

Тестирование API

API - application programming interface, интерфейс прикладного программирования

Тестирование API включает в себя:

- Unit Testing
 - Функциональное тестирование
 - Нагрузочное (Load) тестирование
 - Выявление Runtime Error
 - Security / Fuzz / Penetration тестирование
-
- Тестирование взаимодействия вэб-сервисов (Web Services Interoperability, SOAP)
 - Web UI тестирование

Тестирование API легче автоматизировать, чем тестирование толстых клиентов или клиентов мобильных приложений



Тестирование API

- Информационная совместимость (англ. data compatibility) — способность двух или более систем адекватно воспринимать одинаково представленные данные.
- Программная совместимость (англ. software compatibility) — способность выполнения одинаковых программ с получением одних и тех же результатов. В случае представления программ в виде двоичного кода, говорят о двоичной совместимости.
- Двоичная совместимость, бина́рная совмести́мость (binary compatibility) -- вид программной совместимости, позволяющий программе работать в различных средах без изменения её исполняемых файлов.
- Слом двоичной совместимости -- что безопасно менять в API, а что небезопасно?

ABI (двоичный интерфейс приложений, application binary interface):

- использование регистров процессора
- состав и формат системных вызовов и вызовов одного модуля другим
- формат передачи аргументов и возвращаемого значения при вызове функции

Если интерфейс программирования приложений разных платформ совпадают, код для этих платформ можно компилировать без изменений. Если для разных платформ совпадают и API, и ABI, исполняемые файлы можно переносить на эти платформы без изменений. Если API или ABI платформ отличаются, код требует изменений и повторной компиляции. API не обеспечивает совместимость среды выполнения программы — это задача двоичного интерфейса.

Тестирование API

Краткий субъективный список проверки новой версии API:

- не изменена иерархия классов
 - не удалены существующие классы с публичными методами
 - не удалены публичные методы
 - не изменен тип аргументов или результата публичных методов
 - публичные методы не превращены в inline
 - не добавлена перегрузка методов, где раньше перегрузки не было
 - не изменены параметры const/volatile
 - не добавлены виртуальные методы в классы, где их не было
 - не добавлены override существующих виртуальных методов
-
- не удалены static non-private данные
 - не изменен тип static non-private данных

Инструменты для тестирования API: совпадают со списком для интеграционного, системного интеграционного и юнит-тестирования, перечень небольшой был в прошлой лекции

Можно ли быстро оценить изменения в API без монотонного прогона тестов?

Тестирование API

Статический анализ кода — это процесс выявления ошибок и недочетов в исходном коде программ.


Статический анализ можно рассматривать как автоматизированный процесс обзора кода (code review).

Чем больше проект, тем больше ошибок на 1000 строк кода

Наиболее частое проявление ошибок в API:

- Неопределенное поведение
- Нарушение алгоритма поведения пользователей библиотеки
- Нарушение кроссплатформенности
- Ошибки из-за copy-paste кода

Stage of the bug detection	Cost of fixing
Coding	1
Testing	10
Support	10-25



- Ошибки форматирования строк
- Утечки памяти (код ответственный за утилизацию объектов находится после return)
- Вызов функций не меняющих ничего

Не смотря на существование большого числа инструментов, в большинстве IDE уже есть необходимые средства

Project size (number of code lines)	Typical error density
Less than 2K	0 – 25 errors per 1000 code lines
2K – 16K	0 – 40 errors per 1000 code lines
16K – 64K	0.5 – 50 errors per 1000 code lines
64K – 512K	2 – 70 errors per 1000 code lines
512K and more	4 - 100 errors per 1000 code lines

Тестирование API

Ну и всегда можно сравнить библиотеки бинарно, например с помощью JApiCmp

Ссылка на репорт: https://docs.spring.io/spring-framework/docs/5.1.9.RELEASE_to_5.2.0.RELEASE/spring-core.html

MODIFIED (!) public abstract class org.springframework.util.ReflectionUtils 140

Fields:							
Status	Modifier	Type	Field	Compatibility Changes:			
REMOVED (!)	public static final	org.springframework.util.ReflectionUtils\$MethodFilter (!)		NON_BRIDGED_METHODS			
			Change	Annotations:			
			FIELD_REMOVED	<table><tr><th>Status:</th><th>Fully Qualified Name:</th><th>Elements:</th></tr><tr><td>REMOVED</td><td>java.lang.Deprecated</td><td>n.a.</td></tr></table>	Status:	Fully Qualified Name:	Elements:
Status:	Fully Qualified Name:	Elements:					
REMOVED	java.lang.Deprecated	n.a.					

Methods:							
Status	Modifier	Type	Method	Exceptions	Compatibility Changes:	Line Number	
MODIFIED	static public (<- private)	java.lang.reflect.Method[]	getDeclaredMethods(java.lang.Class)	n.a.	n.a.	Old file	New file
						485	451
NEW	static public	java.lang.reflect.Method[]	getUniqueDeclaredMethods(java.lang.Class, org.springframework.util.ReflectionUtils\$MethodFilter)	n.a.	n.a.	Old file	New file
						n.a.	410
REMOVED (!)	static public	java.lang.Object	invokeJdbcMethod(java.lang.reflect.Method, java.lang.Object)	<div>Status: Name:</div> REMOVED java.sql.SQLException	<div>Change</div> METHOD_REMOVED	Old file	New file
			Annotations:			303	n.a.
			Status: Fully Qualified Name: Elements:				
			REMOVED java.lang.Deprecated			n.a.	
			REMOVED org.springframework.lang.Nullable			n.a.	
REMOVED (!)	static public	java.lang.Object	invokeJdbcMethod(java.lang.reflect.Method, java.lang.Object, java.lang.Object[])	<div>Status: Name:</div> REMOVED java.sql.SQLException	<div>Change</div> METHOD_REMOVED	Old file	New file
			Annotations:			322	n.a.
			Status: Fully Qualified Name: Elements:				
			REMOVED java.lang.Deprecated			n.a.	
			REMOVED org.springframework.lang.Nullable			n.a.	

Динамический анализ кода

Динамический анализ кода - это способ анализа программы непосредственно при ее выполнении. Процесс динамического анализа можно разбить на несколько этапов - подготовка исходных данных, проведение тестового запуска программы и сбор необходимых параметров, анализ полученных данных.

При тестовом запуске исполнение программы может выполняться как на реальном, так и на виртуальном процессоре.

Программы для динамического анализа различаются по способу взаимодействия с проверяемой программой:

- инструментирование исходного кода - в исходный текст приложения, до начала компиляции, добавляется специальный код для обнаружения ошибок;
 - инструментирование объектного кода - код добавляется непосредственно в исполняемый файл;
-
- инструментирование кода на этапе компиляции - проверочный код добавляется, используя специальные ключи компилятора (например, такой режим поддерживается компилятором GNU C/C++ 4.x);
 - не изменяет исходную программу, используются специализированные библиотеки этапа исполнения - для обнаружения ошибок используются специальные отладочные версии системных библиотек

Динамический анализ можно назвать динамическим тестированием, такое тестирование важно там, где главным критерием является надежность программы, время отклика или потребляемые ресурсы.

Чаще всего это тестирование с помощью “серого ящика”.

Пример динамического анализа -- применение стандартных средств JVM для профилирования.

Benchmark на примере Язык-Алгоритмы-Сценарии

Нужно выбрать оптимальный язык, алгоритм, фреймворк и т.д. под условия конкретной задачи – нужен benchmark

Python – один из самых медленных ЯП, Rust – сравним с C

Пример: алгоритмы проверки того, является ли число простым

Пусть есть 3 алгоритма:

- 1) Перебор делителей от $i=2$ до $n-1$ (A1), !!!ТОЛЬКО ДЛЯ СРАВНЕНИЯ МЕДЛЕННОГО И БЫСТРОГО ЯЗЫКА!!!
- 2) Оптимизированный перебор делителей от $i=5$ с исключением кратных 2 или 3 до \sqrt{n} (A2)
- 3) Вероятностный тест Миллера, $\log_2(n)$ – из средств языка, возведение в степень по модулю – самописное (A3)

Пусть есть 2 теста для проверки, какой алгоритм и язык в каких условиях лучше:

- 1) Проверить все числа от 1 до $2^{16}+1$ (это 65537) (T1)
- 2) Проверить все пары чисел от $2^{16}-1, 2^{16}+1$ до $2^{64}-1, 2^{64}+1$ постепенно инкрементируя степень (T2)

Python 3.10.5 (динамическая типизация)

Алгоритм / Тест	A1, мс	A2, мс	A3, мс
T1	11753.021	29.001	2491.975
T2	50654.215	50376.445	137.958

Rustc 1.64.0 + cargo 1.64.0 ($2^{64}+1$ выходит за границы u64, да)

Алгоритм / Тест	A1, мс	A2, мс	A3, мс
T1	5707.151	7.372	663.462
T2	61033.691	3.325	7.105

Benchmark

Выбор например web-фреймворка так же иногда стоит делать на основании требований к производительности в случае конкретной задачи

Примеры сравнения фреймворков для разных языков программирования (включая разные СУБД):

- <https://github.com/the-benchmarkr/website>
- <https://github.com/TechEmpower/FrameworkBenchmarks>

Примеры исследуемых параметров:

- время сериализации JSON / XML
- время обработки единичных запросов
- время обработки пакетов запросов заданного размера
- время проведения транзакций в БД
- скорость передачи информации по шине или другому транспорту
- всё что угодно, что может быть критично для конкретного проекта / приложения / системы

Тестирование производительности

Вид тестирования	Вид тестирования по английский	Вопрос на который отвечает тестирование
Нагрузочное тестирование	Load Testing[2]	Достаточно ли быстро работает система?
Тестирование стабильности	Stability Testing[3]	Достаточно ли надежно работает система на долгом интервале времени?
Тестирование отказоустойчивости	Failover Testing[4]	Сможет ли система переместиться сама на другой сервер в случае сбоя основного сервера?
Тестирование восстановления	Recovery Testing[5]	Как быстро восстановится система?
Стрессовое тестирование	Stress Testing[6]	Что произойдет при незапланированной нагрузке?
Тестирование объемов	Volume Testing[7]	Как будет работать система, если объем базы данных увеличится в 100 раз?
Тестирование масштабируемости	Scalability Testing[8]	Как будет увеличиться нагрузка на компоненты системы при увеличении числа пользователей?
Тестирование потенциальных возможностей	Capacity Testing[9]	Какое количество пользователей может работать?
Конфигурационное тестирование	Configuration Testing[10]	Как заставить систему работать быстрее?

Тестирование производительности

Нагрузочное тестирование

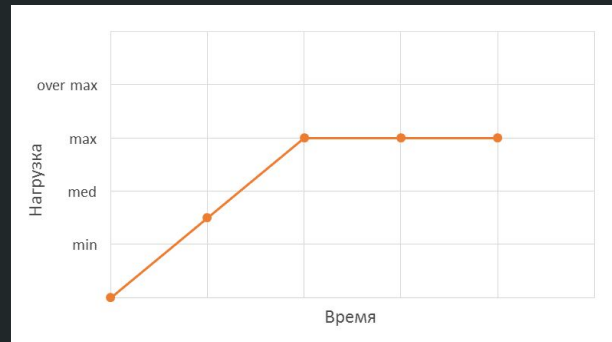
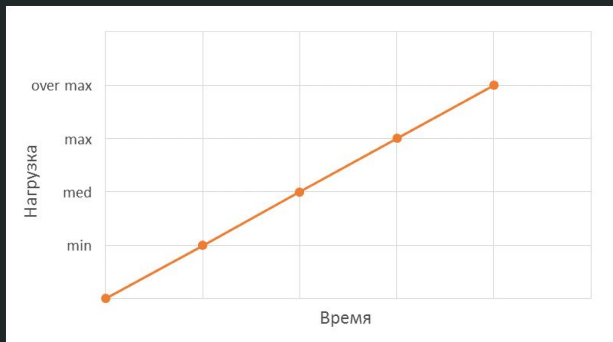
Что оцениваем:

- скоростью работы программного обеспечения
- скоростью работы аппаратного обеспечения
- скоростью работы сети

Как оцениваем:

- измерение времени выполнения выбранных операций при определенных интенсивностях выполнения этих операций
- определение количества пользователей, одновременно работающих с приложением
- определение границ приемлемой производительности при увеличении нагрузки (при увеличении интенсивности выполнения этих операций)

Профиль: определить максимальную рабочую, превысить, оценить работу на максимальной



Тестирование производительности

Тестирование стабильности

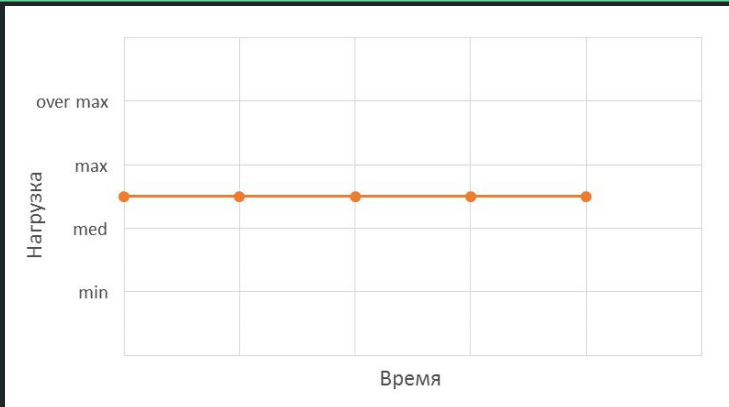
Что оцениваем:

- отсутствие утечек памяти
- отсутствие перезапусков серверов
- отсутствие перезапусков программного обеспечения
- любые ошибки, связанные с накоплением данных
- отсутствие отключений или сбоев в работе сетевого оборудования

Как оцениваем:

- используем средние ожидаемые параметры нагрузки из нагрузочного тестирования

Профиль:



Тестирование производительности

Тестирование отказоустойчивости

Что оцениваем:

- как будет преодолеваться отказ, а именно как система будет перемещать операции между мощностями работающего и нет оборудования
- как будет осуществлен перехват управления системой при отказе управляющего сервера
- как будет осуществлен обход и обработка отказа (переключение на резервный канал связи, отправка данных по другому маршруту и т.д.)

Тестирование восстановления

Что оцениваем:

- время, за которое система восстановится после сбоя
- корректность восстановленных данных

Тестирование производительности

Стрессовое тестирование

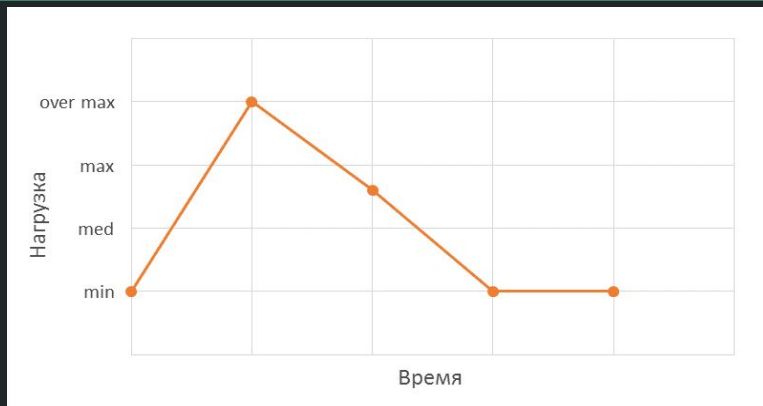
Что оцениваем:

- возможность и время регенерации системы – возможность и время возвращения системы к нормальному состоянию после стрессовых нагрузок
- корректность логирования ошибок и оповещений об их возникновении
- производительность системы при стрессовой нагрузке
- оценка влияния сбоев тестируемой системы на внешние системы

Как оцениваем:

- создаем пиковую нагрузку

Профиль:



Тестирование производительности

Тестирование объемов

Что оцениваем:

- зависимость времени выполнения операций на сервере от объема данных
- количество пользователей, которые могут одновременно работать с приложением “быстро”
- как быстро увеличивается объем данных при работе приложения

Тестирование масштабируемости

Что оцениваем:

- вертикальное масштабирование – увеличения производительности каждого отдельного компонента системы (добавление оперативной памяти на сервере, замена процессора и т.д.) для повышения производительности всей системы в целом
- горизонтальное масштабирование – распределение системы на большее количество серверов параллельно работающих и выполняющих одни и те же функции
- применение временного масштабирования внутри системы с помощью очередей, асинхронных запросов и т.п.

Тестирование потенциальных возможностей - частный случай тестирования масштабируемости, когда оцениваем нагрузку при заранее определенном интервале допустимых значений времени отклика и других параметров

Тестирование производительности

Конфигурационное тестирование

Что оцениваем:

- производительность на разных аппаратных и программных конфигурациях

Как оцениваем:

- создаем пороговую и среднюю нагрузку

Выбираем оптимальную конфигурацию в зависимости от результатов тестирования, требований заказчиков, бюджета проекта

Профиль:

