

Технологии и разработка СУБД

Лекция 5. Практический C/C++

Анастасия Лубенникова
Александр Алексеев

Лекция 5

- Системы сборки
- Тестирование
- Статический анализ
- Профайлеры
- И вот это вот все

Autotools & CMake

- Autotools
 - Стремное легаси
 - К сожалению, все еще много где используется
 - Только *nix
- CMake
 - Добрая магия
 - Стандарт де-факто в современном C/C++
- Есть также SCons и другие

Пример CMakeList.txt

```
cmake_minimum_required(VERSION 3.1)

# так пишутся комментарии

project(project_name)

find_library(PTHREAD_LIBRARY pthread)
find_library(PCRE_LIBRARY pcre)

include_directories(include)
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED on)
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -Wextra -Werror")

add_executable(main src/Main.cpp src/HttpServer.cpp)

target_link_libraries(main ${PTHREAD_LIBRARY} ${PCRE_LIBRARY})
```

Git Submodules

```
$ git submodule add https://github.com/glfw/glfw glfw
```

```
$ git submodule init
```

```
$ git submodule update
```

(И да, Subversion / Mercurial по факту мертвы, не тратьте на них время)

make vs ninja

```
$ mkdir build
```

```
$ cd build
```

```
$ cmake -DCMAKE_BUILD_TYPE=Release -G Ninja ..
```

```
$ ninja -j4
```

```
$ ninja test
```

Форматирование кода: CMakeLists.txt

```
file(GLOB_RECURSE ALL_SOURCE_FILES *.cpp *.h)
```

```
add_custom_target(format
```

```
    COMMAND clang-format --style=file -i ${ALL_SOURCE_FILES} )
```

Форматирование кода: .clang-format

Language: Cpp

AlignEscapedNewlinesLeft: true

AccessModifierOffset: -4

AlignAfterOpenBracket: Align

AlignConsecutiveAssignments: false

AlignConsecutiveDeclarations: false

...

Основные виды тестирования

- Модульное
 - На уровне процедур и классов
- Интеграционное
 - На уровне отдельной библиотеки или микросервиса
- Системное
 - Тестирование всей системы в целом
- Прочие виды
 - Тестирование производительности
 - Тестирование безопасности
 - Тестирование совместимости
 - Юзабилити-тестирование
 - И т.д.

Test-Driven Development (TDD)

Особая крайность, когда сначала пишутся тесты, а потом код. Хорошая практика, как минимум, потому что вы уверены, что тест не проходит, если кода нет или он неправильный. Также вы уверены, что код большей частью покрыт тестами.

Тестирование: CMakeLists.txt

```
enable_testing()
```

```
add_test(NAME python_tests
```

```
    COMMAND py.test -s -v ${CMAKE_SOURCE_DIR}/tests/run.py)
```

Тестирование: PyTest

```
class TestBasic:
```

```
# ...
```

```
def test_index(self):
```

```
    self.log.debug("Running test_index")
```

```
    res = requests.get('http://localhost:{}'.format(PORT))
```

```
    assert(res.status_code == 200)
```

Property-based тесты: пример

```
@pytest.mark.randomize(num=int, min_num=3,  
                        max_num=1000, ncalls=100)
```

```
def test_quickcheck(self, num):
```

```
    result = list(fibgen(num))
```

```
    assert(result[0] < result[-1])
```

```
    assert(len(result) == num)
```

Property-based тесты: зачем?

- Меньше кода, дофига тестов
- Тесты проверяют случаи, о которых вы могли не подумать
- Находится минимальный вход, при котором тест не проходит
- Другие реализации: QuickCheck, ScalaCheck

Code Coverage: CMakeLists.txt

```
option(USE_GCOV "Create a GCov-enabled build." OFF)
```

```
if (USE_GCOV)
```

```
    set(GCC_COVERAGE_COMPILE_FLAGS "-fprofile-arcs -ftest-coverage")
```

```
    set(GCC_COVERAGE_LINK_FLAGS    "-lgcov")
```

```
endif()
```

```
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${GCC_COVERAGE_COMPILE_FLAGS}" )
```

```
set(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} ${GCC_COVERAGE_LINK_FLAGS}" )
```

Code Coverage: генерация отчета

```
cmake -DUSE_GCOV=ON ..
```

```
make
```




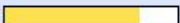
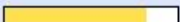

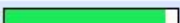




```
make test
```

```
lcov --directory . --capture --output-file summary.info
```

```
mkdir report
```

```
genhtml -o ./report summary.info
```


Code Coverage: результат

Filename	Line Coverage 			Functions 	
HttpRequest.cpp		85.7 %	48 / 56	88.9 %	16 / 18
HttpResponse.cpp		78.3 %	36 / 46	73.3 %	11 / 15
HttpServer.cpp		82.1 %	151 / 184	94.1 %	16 / 17
InMemoryStorage.cpp		0.0 %	0 / 26	0.0 %	0 / 8
Main.cpp		93.0 %	53 / 57	100.0 %	9 / 9
PersistentStorage.cpp		95.9 %	47 / 49	100.0 %	6 / 6
RegexCache.cpp		91.3 %	21 / 23	100.0 %	5 / 5
Socket.cpp		76.6 %	49 / 64	100.0 %	7 / 7
Storage.cpp		0.0 %	0 / 4	0.0 %	0 / 2

Code Coverage: построчно

```
19 :  
20 2 : Status s = rocksdb::DB::Open(options, "hurma_data", &_db);  
21 1 : if(!s.ok())  
22 0 :     throw std::runtime_error("PersistentStorage::PersistentStorage() - DB::Open failed");  
23 1 : }  
24 :  
25 2 : PersistentStorage::~PersistentStorage() {  
26 1 :     if(_db != nullptr)  
27 1 :         delete _db;  
28 1 : }  
29 :  
30 4 : void PersistentStorage::set(const std::string& key, const std::string& value, bool* append) {  
31 :  
32 8 :     std::string json = "{ \"" + key + "\": " + value + " }";  
33 8 :     Document document;  
34 :  
35 4 :     if(!document.Parse(json.c_str()).HasParseError()){  
36 6 :         Status s = _db->Put(WriteOptions(), key, value);  
37 3 :         *append = s.ok();  
38 3 :         if(!s.ok())  
39 0 :             throw std::runtime_error("PersistentStore::set() - _db->Put failed");  
40 :     }  
41 :     else  
42 1 :         *append = false;  
43 4 : }  
44 :
```

Статический анализ кода

- CppCheck
 - Быстрый, но туповатый
 - Бесплатный
- CLang Static Analyzer
 - Достаточно хороший статический анализатор
 - Бесплатный
- PVS-Studio
 - Тоже хороший анализатор
 - За деньги
- Coverity Scan
 - Считается лучшим
 - Очень дорого
 - Есть веб-версия, бесплатная для открытых проектов

Статический анализ: Clang Static Analyzer

```
$ cmake -DCMAKE_C_COMPILER=`which clang` \
```

```
-DCMAKE_CXX_COMPILER=`which clang++` -G Ninja ..
```

```
$ ninja clean
```

```
$ mkdir -p ~/temp/report
```

```
$ scan-build -o ~/temp/report ninja -j4
```

Статический анализ: пример отчета

```
1573      /* Copy SubXIDs, if present. */
1574      if (serialized_snapshot->subxcnt > 0)
1575      {
1576          snapshot->subxip = snapshot->xip + serialized_snapshot->xcnt;
1577          memcpy(snapshot->subxip, serialized_xids + serialized_snapshot->xcnt,
1578                serialized_snapshot->subxcnt * sizeof(TransactionId));
1579      }
```

2 ← Taking true branch →

3 ← Null pointer value stored to field 'subxip' →

4 ← Null pointer argument in call to memory copy function

Valgrind: зачем?

- Находит утечки памяти
- Находит обращения к неинициализированной памяти
- В отличие от аналогов (MemorySanitizer) стабилен, гибок, работает везде
- Из минусов: замедляет выполнение программы в 10-20 раз

Valgrind: пример использования

```
$ gcc -O0 -g vgcheck.c -o vgcheck
```

```
$ valgrind --leak-check=full --track-origins=yes ./vgcheck
```

Valgrind: пример отчета

==2205== Conditional jump or move depends on uninitialised value(s)

==2205== at 0x4E800EE: vfprintf (in /usr/lib/libc-2.25.so)

==2205== by 0x4E87EA5: printf (in /usr/lib/libc-2.25.so)

==2205== by 0x4005CA: run_test (vgcheck.c:10)

==2205== by 0x4005F4: main (vgcheck.c:18)

==2205== Uninitialised value was created by a stack allocation

==2205== at 0x400586: run_test (vgcheck.c:6)

Valgrind: что в нем еще есть

- Флаг `--suppressions`
- Callgrind - профилировщик
- Massif - профилировщик памяти
- Helgrind - инструмент поиска состояний гонки

Heaptrack: профилировщик памяти

```
$ heaptrack ./test_rbtrees
```

```
# или: heaptrack -p PID
```

```
$ heaptrack_print --print-leaks \
```

```
--print-histogram histogram.data \
```

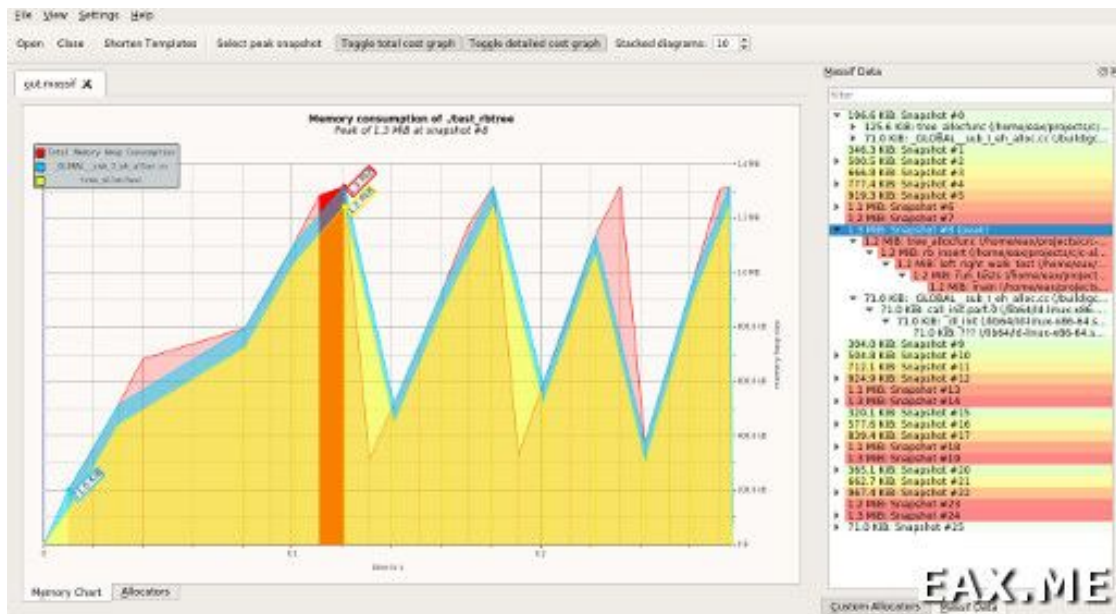
```
--print-massif massif.data \
```

```
--print-flamegraph flamegraph.data \
```

```
--file ./heaptrack.test_rbtrees.22023.gz > report.txt
```

Heartrack: визуализация собранных данных

\$ massif-visualizer massif.data



Как не нужно делать бенчмарки

- Неповторяемость
- Вы измеряете не то, что думаете
- Взятие среднего
- Кто будет бенчмаркать бенчмарки?
- Отсутствие анализа (правка наугад)
- Игнорирование ошибок
- Нетипичная нагрузка
- Маркетинг и подгон
- А как же другие параметры?

perf top

```
$ sudo perf top -p 12345
```

```
Samples: 3M of event 'cycles', Event count (approx.): 235
32.80% postgres [.] list_nth
20.29% postgres [.] ResourceOwnerForgetRelationRef
12.87% postgres [.] find_all_inheritors
 7.90% postgres [.] get_tabstat_entry
 6.68% postgres [.] ResourceOwnerForgetTupleDesc
 1.17% postgres [.] hash_search_with_hash_value
 0.84% postgres [.] SearchCatCache
 0.65% postgres [.] AllocSetAlloc
 0.43% [kernel] [k] clear_page_c_e
 0.42% libc-2.19.so [.] lseek64
 0.36% postgres [.] ExecInitExpr
```

perf record

```
$ sudo perf record -p 12345 -F 99 -g
```

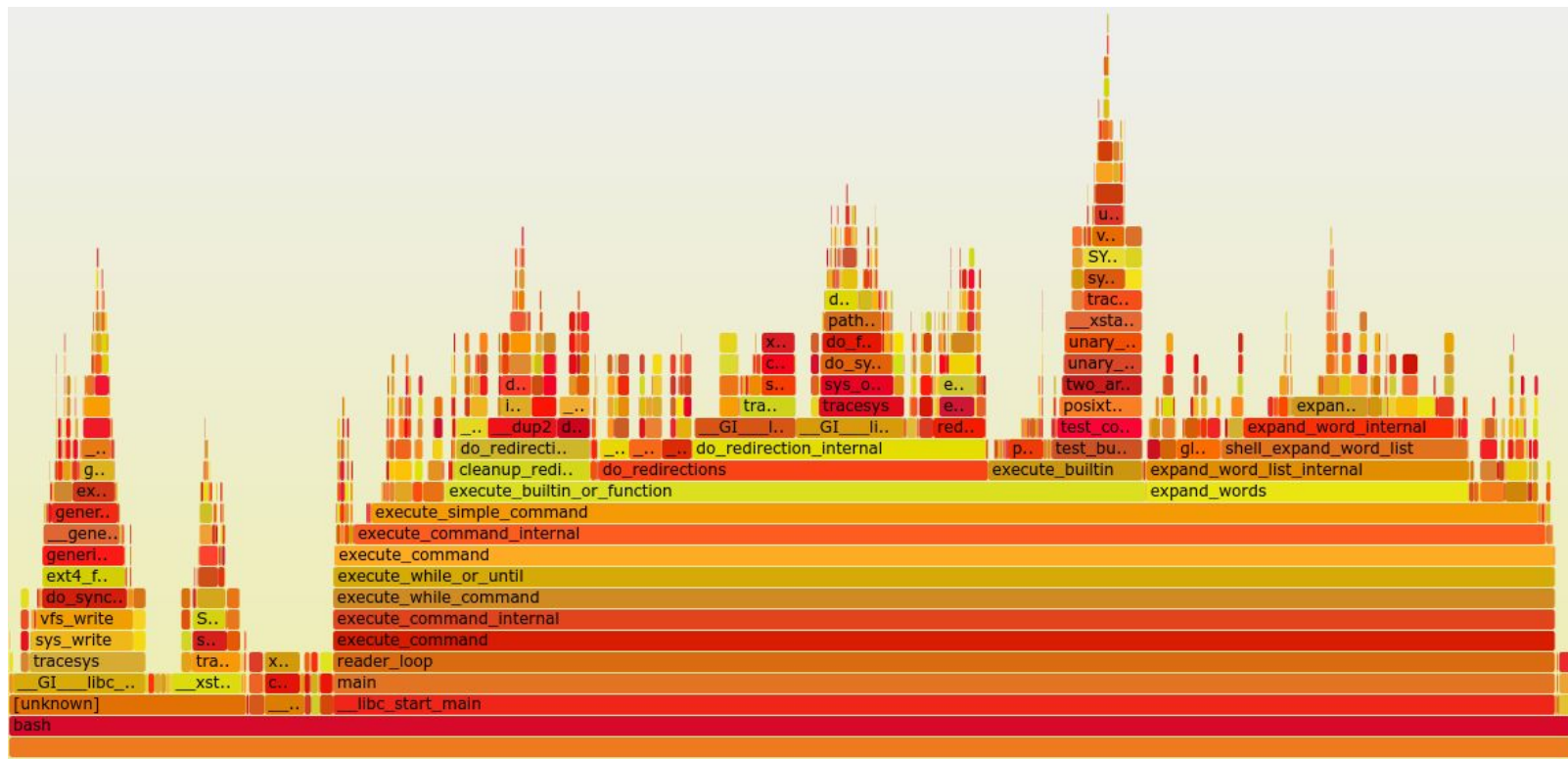
```
$ git clone https://github.com/brendangregg/FlameGraph
```

```
$ sudo perf script | \
```

```
    ./FlameGraph/stackcollapse-perf.pl > out.perf-folded
```

```
$ ./FlameGraph/flamegraph.pl out.perf-folded > perf.svg
```

Получаем флеймграф



Один полезный прием с GDB

```
$ gdb --batch --command=gdb.script -p 12345
```

... где в `gdb.script` написано просто:

```
bt
```

Иногда позволяет найти, где процесс долго висит в блокировке.

DTrace и компания

Инструменты трассировки ядра операционной системы и пользовательских приложений с целью их профайлинга и отладки (еще иногда для security-аудита).

- DTrace
 - FreeBSD, MacOS. Есть dtrace4linux.
 - См также DTrace Toolkit: opensnoop, execsnoop и тд
- SystemTap
 - Linux
 - Больше как инструмент разработчиков ядра, страшно тащить в продакшн.
- bcc/eBPF
 - Добрая магия в Linux 4.1+ (лучше 4.9+)

DTrace: пример профилирования

```
$ sudo dtrace \
```

```
-n 'profile-4999 /execname == "postgres"/ {@[ustack(1)] = count()}'
```

```
$ sudo dtrace \
```

```
-n 'profile-4999 /pid == 1234/ { @[ustack()] = count() }' \
```

```
-o out.dtrace
```

DTrace: строим флеймграф

```
$ git clone https://github.com/brendangregg/FlameGraph
```

```
$ perl ./FlameGraph/stackcollapse.pl out.dtrace > out_folded.dtrace
```

```
$ perl ./FlameGraph/flamegraph.pl ./out_folded.dtrace > fg.svg
```

Флеймграфы с bcc/eBPF

```
$ sudo /usr/share/bcc/tools/profile -df -p 32133 > out.profile
```

```
$ git clone https://github.com/brendangregg/FlameGraph
```

```
$ ./FlameGraph/flamegraph.pl --colors hot < out.profile > out.svg
```

Полезняшки из bcc-tools

- execsnoop
- opensnoop
- biotop
- biolatancy
- tcplife
- tcptop
- cpudist
- filetop
- gethostlatency
- trace
- argdist

Что осталось за кадром

- В чем писать код
 - Vim, Sublime Text, CLion, тысячи их
- Документация
 - Markdown, можно сказать, стандарт
 - Doxygen скорее стоит посмотреть, чем нет
- Отладчики
 - GDB, LLDB, WinDBG + те, что в IDE
 - reverse debugging, например с RR
- Ассемблер и дизассемблеры
 - Стоит знать как минимум x86/x64
 - Инструменты: objdump, Hopper, IDA Pro + gdb/lldb
- CLang-санитайзеры
 - MemorySanitizer, ThreadSanitizer, etc

Дополнительные материалы

- “Systems Performance: Enterprise and the Cloud” by Brendan Gregg
- “21st Century C: C Tips from the New School” by Ben Klemens
- “Modern X86 Assembly Language Programming” by Daniel Kusswurm
- “ScalaCheck: The Definitive Guide” by Rickard Nilsson
- <http://brendangregg.com/blog/index.html>
- <https://github.com/afiskon/hurmadb>
- <https://eax.me/tag/c-cpp/>

Домашние задания

BRACE YOURSELF

DEADLINE IS COMING

 memegenerator.com

Вопросы и ответы.

- a.lubennikova@postgrespro.ru
- a.alekseev@postgrespro.ru
- Telegram: <https://t.me/dbmsdev>