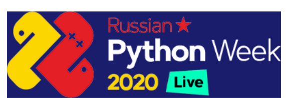
Conan и Python. Как работает современный DevOps для C++

Евгений Селиверстов

15 сентября 2020 г.



О себе

Евгений Селиверстов

Найти в интернете — **@theirix**

Код — https://github.com/theirix

Что делаю:

- Пишу на C++, Python, Ruby и других языках более 15 лет
- Исследовательская работа с CUDA, scipy
- CTO в Агро Сотфвер
- Люблю опен-сорс
- Поддерживаю пакеты в Debian, Homebrew, Pypi, Conan
- Контрибьютор в Conan, Conan Center, Bincrafters

Содержание

1 Особенности DevOps для C++

2 Python как язык рецептов

3 Python как язык инфраструктуры

4 Устройство и развитие открытой разработки

Содержание

1 Особенности DevOps для C++

2 Python как язык рецептов

3 Python как язык инфраструктуры

4 Устройство и развитие открытой разработки

Особенности DevOps для C++

- Сложно.

Особенности DevOps для C++

- Сложно.
- В мире C/C++ нет стандартного способа сборки, пакетирования и распространения пакетов.
- В Python, Ruby, Rust, Java, Go, Haskell — есть. Легче вход, меньше затрат на велосипедостроение.
- Пакетирование перекладывается с разработчиков библиотек на мейнтейнеров и девопсов.
- Сложности с кросс-платформенной сборкой.

Особенности сборки и пакетирования C++

Проблемы:

- нет единой системы сборки
- нет пакетов
- нет способа описать зависимости
- нет способа вытащить зависимости (GitHub, SVN, HTTP)
- неочевидно версионирование
- различающийся бинарный интерфейс (ABI)

Особенности сборки и пакетирования C++

Проблемы:

- нет единой системы сборки
- нет пакетов
- нет способа описать зависимости
- нет способа вытащить зависимости (GitHub, SVN, HTTP)
- неочевидно версионирование
- различающийся бинарный интерфейс (ABI)

Типовые используемые решения:

- системные библиотеки
- развертывание скриптами
- git submodule
- решения от билд-систем (CMake ExternalProject)
- monorepo - Bazel, Buck

Что такое Conan

Conan - это кроссплатформенный децентрализованный менеджер зависимостей и пакетов для C++.

Since 2015^{TM}

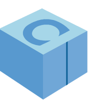
https://conan.io

https://github.com/conan-io/conan

Преимущества

Универсальность и гибкость.

- платформы - Linux, macOS, Windows, UNIX
- таргеты - нативные, Android, iOS, emscripten
- системы сборки - CMake, MSBuild, Autotools, Make, Xcode
- компиляторы - gcc, clang, msvc, intel compiler
- бинарные пакеты и исходные коды
- децентрализованность
- открытый код, развитие и разработка
- гибкость в рецептах и расширениях - спасибо, Python!



Менеджеры зависимостей и пакетные менеджеры

Менеджер зависимостей:

- построение графа зависимостей с ограничениями
- управление версиями пакетов
- работа на уровне исходных кодов и манифестов

Менеджеры зависимостей и пакетные менеджеры

Менеджер зависимостей:

- построение графа зависимостей с ограничениями
- управление версиями пакетов
- работа на уровне исходных кодов и манифестов

Пакетный менеджер:

- сборка бинарных пакетов
- скачивание и публикация бинарных пакетов
- разделение пакетов между проектами

Что такое Conan

Чем является Conan:

- менеджеров зависимостей
- пакетным менеджером

Что такое Conan

Чем является Conan:

- менеджеров зависимостей
- пакетным менеджером

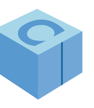
Чем не является Conan:

- системой сборки
- системным пакетном менеджером (apt, dnf)
- системой установки ПО (homebrew, nuget)

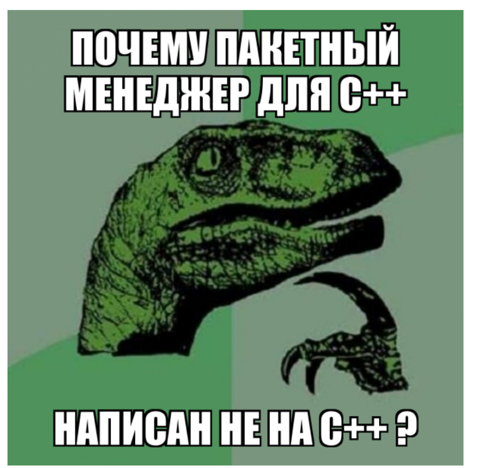
Что такое пакет Conan

Пакет является единицей сборки, загрузки и выгрузки с репозитариев.

- рецепт — файл conanfile.py с описанием пакета
- метаданные — опции сборки, версия компилятора
- бинарный артефакт — скомпилированные бинарники, заголовочные файлы
- манифест с хэш-суммами
- лицензия



Почему для Conan выбран python



Почему для Conan выбран python

Причины:

- возможность установить на всех платформах, где есть Python
- рецепты пакетов - DSL на Python
- легкая расширяемость пакетного менеджера в рантайме

Почему для Conan выбран python

Причины:

- возможность установить на всех платформах, где есть Python
- рецепты пакетов - DSL на Python
- легкая расширяемость пакетного менеджера в рантайме

В C++ всё это сделать сложнее:

- как собрать пакетный менеджер без пакетного менеджера?
- сложнее кроссплатформенная сеть
- сложнее кроссплатформенное шифрование

Содержание

1 Особенности DevOps для C++

2 Python как язык рецептов

3 Python как язык инфраструктуры

4 Устройство и развитие открытой разработки

Python как язык рецептов

Рецепт `conanfile.py` включает в себя все описания источников, опций, инструкций сборки и пакетирования.

Conanfile похож по смыслу на `setup.py` - сочетание декларативного описания и кода.

Python как язык рецептов

Рецепт `conanfile.py` включает в себя все описания источников, опций, инструкций сборки и пакетирования.

Conanfile похож по смыслу на `setup.py` - сочетание декларативного описания и кода.

В conanfile возможны:

- императивные инструкции (последовательность команд)
- использование DSL-like helpers
- плавный переход к декларативному описанию - `conandata.yml`
- любой код на Python!

Пример conanfile.py (I)

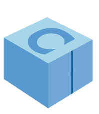
```python
from conans import ConanFile, CMake, tools

class Bzip2Conan(ConanFile):
  name = "bzip2"
  version = "1.0.8"
  settings = "os", "compiler", "arch", "build_type"
  options = {"shared": [True,False], "fPIC": [True,False],
             "build_executable": [True,False]}
  default_options = {"shared": False, "fPIC": True,
                     "build_executable": True}
  exports_sources = "CMakeLists.txt"
  generators = "cmake"

  def source(self):
    tools.get("https://sourceware.org/pub/bzip2/" +
      "%s-%s.tar.gz" % (self.name, self.version))
    os.rename("%s-%s" % (self.name, self.version), "src")
  ...
```

Пример conanfile.py (II)

```
1 ...
2
3 def _configure_cmake(self):
4   cmake = CMake(self)
5   cmake.definitions["BZ2_BUILD_EXE"] = \
6     self.options.build_executable
7   cmake.configure()
8   return cmake
9
10 def build(self):
11   self._configure_cmake().build()
12   cmake.build()
13
14 def package(self):
15   self.copy("LICENSE", dst="licenses", src="src")
16   self._configure_cmake().install()
17
18 def package_info(self):
19   self.cpp_info.libs = tools.collect_libs(self)
```



Методы в conanfile - source

Метод `source` — достать исходный код апстримного пакета.

Вариант с явным вызовом команды:

```
1 def source(self):
2   self.run("git clone " +
3       "https://gitlab.com/federicomenaquintero/bzip2.git")
```

Методы в conanfile - source

Вариант с использование вспомогательных функций Python.

Пакет conans.tools поставляется с Conan.

```
1 from conans import tools
2
3 version = "1.0.8"
4
5 def source(self):
6     tools.get("https://sourceware.org/pub/bzip2/" +
7      "%s-%s.tar.gz" % (self.name, self.version))
```

Методы в conanfile - source

Развязываем код и метаданные.

Использование метаданных вне рецепта в `conandata.yml`, которые Conan автоматически помещает в `self.conan_data`.

```
1 def source(self):
2   tools.get(self.conan_data["sources"][self.version])
```

Содержимое conandata.yml

```
1 sources:
2   1.0.8:
3     url: "https://sourceware.org/pub/bzip2/bzip2-1.0.8.tar.gz"
4     sha256: ab5a03176ee106d3f0fa90e381da478ddae405918153cca248e682cd0c4a2269
```

Conanfile - скриптовый метод build

Conan - не билд-система. Используем поддерживаемую апстримным пакетом.

Conanfile - скриптовый метод build

Conan - не билд-система. Используем поддерживаемую апстримным пакетом.

Наивный подход:

```python
def build(self):
    self.run("./configure CFLAGS='-DMAGIC=NO -fPIC' " +
             "--enable-shared --disable-static " +
             "--prefix=%s" % self.package_folder)
    self.run("make")
    self.run("make install")
```

Conanfile - скриптовый метод build

Conan - не билд-система. Используем поддерживаемую апстримным пакетом.

Наивный подход:

```
1 def build(self):
2   self.run("./configure CFLAGS='-DMAGIC=NO -fPIC' " +
3            "--enable-shared --disable-static " +
4            "--prefix=%s" % self.package_folder)
5   self.run("make")
6   self.run("make install")
```

Минусы:

- императивно
- негибко
- проброс переменных в скрипт неудобен

Можно ли лучше?

Conanfile - метод build с хелперами

Хелперы из пакета Conan!

```
1 from conans import AutoToolsBuildEnvironment
2
3 def build(self):
4   autotools = AutoToolsBuildEnvironment(self)
5   autotools.defines.append("MAGIC=YES")
6   autotools.configure()
7   autotools.make()
8   autotools.install()
```

Conanfile - метод build с хелперами

Пример для CMake:

```
1 from conans import CMake
2
3 def build(self):
4   cmake = CMake(self)
5   cmake.definitions["MAGIC"] = True
6   cmake.definitions["WITH_FOO"] = self.options.with_foo
7   cmake.configure()
8   cmake.make()
9   cmake.install()
```

Плюсы:

- Объектная модель
- Автоматическое управление путями
- Высокоуровневое управление опциями fPIC, shared

Conanfile - метод build с хелперами

Пример для MSBuild:

```
1 from conans import MSBuild
2
3 def build(self):
4   msbuild = MSBuild(self)
5   msbuild.build(project_file="bzip2.sln", targets=["bz2"])
```

Conanfile - метод package

Ручная выборка файлов:

```
1 def package(self):
2   self.copy(pattern="COPYING", dst="licenses")
3   self.copy(pattern="*.so*", dst="lib")
```

Использование инсталляции в директорию пакета:

```
1 def package(self):
2   self.copy("LICENSE", dst="licenses")
3   self._configure_cmake().install()
4   tools.rmdir(os.path.join(self.package_folder, "docs"))
```

Далеко не всё из инсталлированных файлов нужно (docs, utils).

В пакете оказываются строго необходимые файлы, лицензии.

Conanfile - опции

Пакеты C++ без опций бесполезны. Каждый бинарный пакет определен именем, версией, набором опцией и набором настроек.

Опции - булевы, строковые или предметные переменные.

```
1 options = {
2   "shared": [True, False],
3   "with_zlib": [True, False],
4   "crypto_backend": ["openssl", "mbed"]}
5
6 default_options = {
7   "shared": False,
8   "with_zlib": True,
9   "crypto": "openssl"}
```

Conanfile - тонкая настройка опций

Опции могут быть ограничены или настроены в коде исходя из:

- других опций
- версии пакета
- версии компилятора
- целевой ОС

```
1  def configure(self):
2    if self.options.shared and self.options.crypto == "tls":
3      raise ConanInvalidConfiguration("Bad options")
4
5    if self.settings.os == "Windows":
6      self.options.with_zlib = False
7
8    if self.settings.compiler == "gcc" and \
9        self.settings.compiler.version <= Version(3):
10     raise ConanInvalidConfiguration("Ancient gcc")
```

Conanfile - зависимости

Зависимости пакета указываются и настраиваются внутри Conanfile

```
1 def requirements(self):
2   self.requires("openssl/1.1.1g")
3   if self.require_foobar
4     self.requires("foobar/0.1@myrepo/testing")
5   self.requires("zlib/1.2.11")
```

В зависимости указывается имя и версия, а также пользователь (myrepo) и канал (testing).

По умолчанию пакеты находятся в центральном репозитарии Conan Center.

Другие возможности

Рецепт может включать многое другое:

- метаданные (URL, автор, лицензия)
- патчи исходного кода
- `package_info` - метаданные для билд-систем
- `configure, config_options` - настройки опций пакета
- `imports` - необходимые файлы к началу сборки
- `requirements` - зависимости на другие пакеты
- `build_requirements` - зависимости на билд-тулы (protoc, ninja)
- `exports` - экспортируемые в пакет файлы, кроме самого рецепта

Путь к декларативному описанию

Общая идея рецептов:

- краткость для простых пакетов
- гибкость для сложных пакетов

Многие методы для простоты могут быть упрощены до поля в классе: `requires = ("zlib/1.2.11")`

Изменение исходных кодов выделяется в набор патчей в `conandata.yml`.

Формат описания зависимостей

Описывает зависимости клиента (приложение или инсталляция).

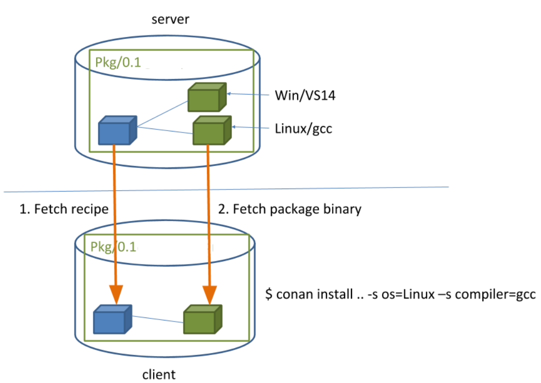
```
1 [requires]
2 poco/1.9.4
3 libcurl/7.56.1
4
5 [generators]
6 cmake
7
8 [options]
9 poco:enable_crypto=True
```

По смыслу похож на requirements.txt

Установка пакета

```
conan install .
```

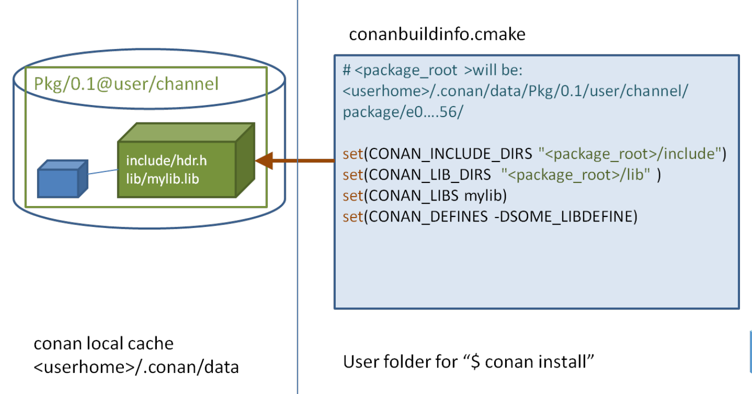
Conan разрешает граф зависимостей, загружает и устанавливает бинарные пакеты. Если в репозитарии нет бинарного пакета, происходит создание пакетов локально.



Потребление пакета

В зависимости от выбранного генератора создаются файлы интеграции с билд-системой.

Для CMake создается `conanbuildinfo.cmake` с переменными, указывающими на локальный кэш с пакетами.



Потребление пакета

Подключение в CMake потребляющего проекта:

```
1 include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
2 conan_basic_setup()
3 ...
4 target_link_libraries(example ${CONAN_LIBS})
```

Для современного компонентного CMake:

```
1 ...
2 target_link_libraries(example CONAN_PKG::poco)
```

Создание пакета

Основная функциональность пакетного менеджера.

Вход: рецепт.

Выход: бинарный пакет с метаданными.

```
conan create . poco/1.9.4 -o poco:enable_crypto=True
```

Под капотом - как интерпретируется рецепт

При вызове команд Conan выполняет импорт conanfile.py при помощи `importlib`.

Стадии сборки — source, build, package.

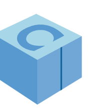
Методы каждой стадии выполняются независимо над своим окружением (поля экземпляра Conanfile, переменные окружения).

Резолвер зависимостей

- Все опции генерируются и хранятся в пакете рядом с `conanfile.py` в экспортном формате `conaninfo.txt`.
- Резолвер зависимостей быстрый.

Резолвер зависимостей

- Все опции генерируются и хранятся в пакете рядом с `conanfile.py` в экспортном формате `conaninfo.txt`.
- Резолвер зависимостей быстрый.
- Сравним его с pip, pip-tools, poetry, pipenv.
  Необходима полная интерпретация `setup.py` для разрешения графа зависимости.



Резолвер зависимостей

Данные `package_info` для корректной сборки против зависимостей. Требуют интерпретации рецепта по importlib.

Upstream - данные, предоставляемые зависимостями:

```
1 def package_info(self):
2   self.cpp_info.includedirs = ["include/cool"]
3   self.cpp_info.libs = ["libcool"]
4   self.cpp_info.defines= ["DEFINE_COOL=1"]
```

Downstream - потребление данных от зависимостей:

```
1 def build(self):
2   print(self.deps_cpp_info["bzip2"].include_paths)
3   print(self.deps_cpp_info["openssl"].sharedlinkflags)
4   print(self.deps_cpp_info["openssl"].version)
```

Расширение рецептов при помощи Python

Если не хватает встроенной функциональности.

Расширение рецептов при помощи Python

Если не хватает встроенной функциональности.

Механизм python_requires.

Создаем псевдо-пакет:

```
1  class MyConanBase(object):
2    def build(self):
3      self.output.info("build code with magic")
4
5    def package(self):
6      self.output.info("package code with magic")
7
8  class PyReq(ConanFile):
9    name = "pyreq"
10   version = "0.1"
```

Расширение рецептов при помощи Python

Используем псевдо-пакет pyreq в нашем рецепте:

```
1 from conans import ConanFile
2
3 class Bzip2Conan(ConanFile):
4     python_requires = "pyreq/0.1@user/channel"
5     python_requires_extend = "pyreq.MyBase"
6
7   def source(self):
8     tools.get(self.conan_data["sources"][self.version])
```

Собираем:

`conan create .`

Пакет собирается сам с поведением из базового класса MyBase.

Под капотом - как работает расширение

- Интерпретатор загружает conanfile.py из пакетов, перечисленных в `python_requires`, через `importlib`.
- Пакеты с расширениями должны экспортировать только свой код.
- Все методы и поля расширения доступны в рецепте через `self.python_requires["pyreq"].module`

Виртуальные окружения

Пакеты устанавливают бинарные артифакты в свои директории:

`~/.conan/data/package/version/.../bin`,
`~/.conan/data/package/version/.../lib`

- Установка переменных `PATH`, `LD_LIBRARY_PATH`.
- Rpath
- Запуск программы-потребителя неудобен.

Виртуальные окружения

Решение из мира Python — virtualenv.

Запуск окружения:

```
1 conan install . -g virtualenv
2 source activate.sh
3 echo $PATH
4 mypackage-cli --help
```

Содержание

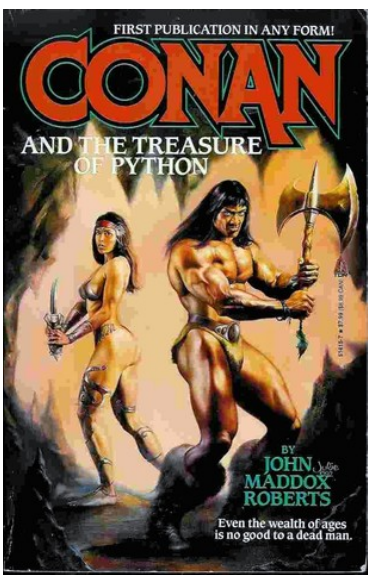
1 Особенности DevOps для C++

2 Python как язык рецептов

3 Python как язык инфраструктуры

4 Устройство и развитие открытой разработки

Python как язык инфраструктуры



Клиент-серверная архитектура Conan

Conan состоит из:

- клиентской программы conan
- открытого протокола REST
- сервера Conan

Клиент-серверная архитектура Conan

Conan состоит из:

- клиентской программы conan
- открытого протокола REST
- сервера Conan

Remote (назначение) - место для загрузки и выгрузки пакетов.

Децентрализованность (как и в случае pypi).

Как построить пайплайн сборки

Задачи:

- достать и опубликовать зависимости — это к серверам
- собрать финальные продукты — это к продуктовым сборкам
- собрать пакеты с библиотеками — задача open-source community

Как делается сборка открытых библиотек?

Сборка множественных конфигураций

Сборка открытых библиотек — неизвестен потребитель.

Многомерность конфигурации:

- settings.os - Linux, macOS, Windows, FreeBSD, Emscripten
- settings.compiler - gcc, clang, msvc, apple-clang
- settings.compiler.version - `[[:digit:]]+`
- settings.compiler.libcxx - libc++, libstdc++
- settings.compiler.runtime = MT, MD, MTd
- options.shared = True, False
- options.with_openssl = True, False
- ...

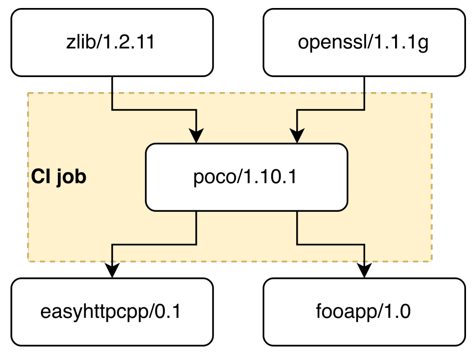
Гиперкуб конфигураций

Нужно:

- Ограничения на типовые конфигурации и сочетаемость.
- Инструментарий для распределения конфигураций на CI.

Модель сборки - граф зависимостей

Сборка пакета poco.



Модель сборки - смешанный билд

```
conan create --build=missing poco/1.10.1
```

Conan Repository

poco sources

poco recipe

func( )

deps sources

func( )

poco/1.10.1

CI job

poco binary package

Модель сборки - чистый бинарный билд

```
conan create --build=poco poco/1.10.1
```

Conan Repository

deps binary packages

poco sources

poco recipe

func( )

func( )

poco/1.10.1

CI job

poco binary package

Модель сборки библиотек Conan

- Все зависимости по графу должны быть уже собраны
- Транзитивные зависимости
- Воспроизводимые билды

Модель сборки библиотек Conan

- Все зависимости по графу должны быть уже собраны
- Транзитивные зависимости
- Воспроизводимые билды

Как соотнести джобу CI и сборку одной конфигурации?

Python-пакеты инфраструктуры Conan

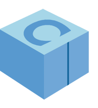
Пакет conan_package_tools — сборка множественных конфигураций

Пакет bincrafters_package_tools — интеграция CPT для сборки пакетов комьюнити Bincrafters

Пакет bincrafters_conventions — линтер рецептов Bincrafters

Как эволюционировала инфраструктура Conan

1. Собственные репозитарии на bintray. Нет общего CI. Сборка ведется автором рецепта, часто локально.
2. Появление репозитария Conan Center на bintray. Рецензируемое принятие в remote conan center.
3. Появление комьюнити Bincrafters. Сборка рецептов на открытых платформах CI (Travis, Appveyor, CircleCI).
4. Автоматизация и шаблонизация сборок Bincrafters. Репозитарий на пакет, бранч на версию.
5. Появление Conan Center Index. Единый репозитарий рецептов. Сборка на закрытом Jenkins на мощностях JFrog.



Пайплайн с нуля

`ci-run.sh`:

```
1 conan create . poco/1.9.4@user/testing \
2   -s compiler=apple-clang -s compiler.version=11.0 \
3   -o poco:shared=True -o poco:with_openssl=False
4 conan upload poco/1.9.4@user/testing \
5   --all --remote my-bintray-repo
```

Повторить для всех конфигураций (ОС, компиляторы, опции).

Пайплайны с conan_package_tools на CI

Пусть конфигурации построит скрипт на Python.

`ci-run.sh`:

```
1 pip install conan conan_package_tools
2 python build.py
```

`build.py`:

```
1 from conan.packager import ConanMultiPackager
2
3 builder = ConanMultiPackager()
4 builder.add_common_builds("poco:shared", pure_c=False)
5 builder.run()
```

Скрипт перечисляет в add_common_builds только те конфигурации, которые ограничены переменными окружения.

Пайплайны с conan_package_tools

Запустим сборку локально (macOS):

```
1 export CONAN_ARCHS=x86_64,x86
2 export CONAN_APPLE_CLANG_VERSIONS=7.3,10.0
3 export CONAN_GCC_VERSIONS=5,7
4 export CONAN_BUILD_TYPES=Debug
5 python build.py
```

Исполнитель - нативный процесс (Xcode) или Docker-контейнер с GCC.

Автоматическая публикация пакетов в CONAN_UPLOAD.

Пайплайны с conan_package_tools

Запускаем скрипт в каждой ячейке матричной джобы с переменными окружения.

.travis.yml:

```
1  - <<: *linux
2    env: CONAN_GCC_VERSIONS=5 CONAN_DOCKER_IMAGE=conanio/gcc5
3  - <<: *linux
4    env: CONAN_GCC_VERSIONS=7 CONAN_DOCKER_IMAGE=conanio/gcc7
5  - <<: *osx
6    osx_image: xcode7.3
7    env: CONAN_APPLE_CLANG_VERSIONS=7.3 CONAN_ARCHS=x86,x86_64
8  - <<: *osx
9    osx_image: xcode10.1
10   env: CONAN_APPLE_CLANG_VERSIONS=10.0
11 ...
12 script:
13   - ./cirun.sh
```

Распараллеливание на уровне одной джобы CI.

Тонкая настройка ограничений с conan_package_tools

- Несовместимые конфигурации.
- Избыточные конфигурации.

build.py:

```
1  builder = ConanMultiPackager()
2  builds = builder.add_common_builds("poco:shared",
3                                     pure_c=False)
4
5  filtered = []
6  for settings, options, e, br, ref in builder.items:
7    # exclude gcc shared builds for some reason
8    if settings["compiler"] == "gcc" and \
9          options["poco:shared"] is True:
10     continue
11   filtered.append([settings, options, e, br, ref])
12 builder.builds = filtered
13
14 builder.run()
```

Пайплайны bincrafters

Комьюнити Bincrafters.

Инкубация и подготовка пакетов для приёма в Conan Center (2017-2019).

Инфраструктура на Python.

Пакет bincrafters_package_tools - улучшение conan_package_tools:

- структура бранчей задает версии и каналы - testing/1.2.3
- автоматическое назначение множества переменных CPT
- генерация конфигураций CI - Travis, Appveyor, CircleCI
- настройка новых репозитариев GitHub с токенами CI

Результат - резкое ускорение прототипирования и пакетирования новых библиотек.

Пайплайны conan-center-index

Monorepo meets Conan

Подход:

- все рецепты в одном бранче (master)
- разделение на версии внутри одной директории (либо all для универсального рецерта)
- специфичные для конкретных версий данные (URL, хэшсуммы, патчи) хранятся в метаданных `conandata.yml`

Пример: `recipes/bzip2/all/conanfile.py`, `recipes/bzip2/1.0.9/conanfile.py`

Пайплайны conan-center-index

Conan-Center Continuous Integration (C3I)

Плюсы:

- код рецептов развязан с метаданными
- упрощается рефакторинг
- нет управления сотнями репозитариев

Минусы:

- специфический CI, учитывающий уже собранные пакеты

Docker для сборки пакетов

Большинство рецептов собираются в Docker.

Исключения: Xcode и Visual Studio - либо разные сборочные машины, либо несколько компиляторов на одной машине

conan_docker_tools - коллекция Docker-образов с предустановленными компиляторами и Conan.

Проблема выбора базового образа Linux:

- достаточно старая версия glibc для обратной совместимости ABI
- установка компиляторов gcc, clang
- установка Python 3
- multi-stage образы Docker

Автоматизация - хуки

Хук — код на Python, используемый Conan для расширения клиента.

Может быть единым скриптом или целым пакетом.

Работает над объектной моделью Conanfile.

Точки интеграции:

- pre_export
- pre_download
- post_build
- post_upload_package
- ...

Похоже на git hooks.

Автоматизация - хуки. Пример

~/.conan/hooks/license_checker.py:

```
1 def pre_export(output, conanfile, *args, **kwargs):
2   if not getattr(conanfile, "license", None):
3     output.warn("No license")
```

Автоматизация - известные хуки

Хуки Conan по назначению:

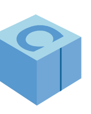
- Локальные хуки при локальном экспорте пакета
- Боты-линтеры для PR на github
- Боты для наполнения метаданных GitHub, Bintray

Стандартизация линтера — имена конвенций вида KB-H019.

Хуки bincrafters_conventions — независимый от hooks механизм. Включают линтеры и автоматические исправления. Хорошо работает на модели Bincrafters множественных репозитариев.

Автоматизация - линтеры

- Проверка корректности Python рецептов - pylint.
- Проверка семантики рецептов.
- Проверка корректности бинарных пакетов.
- Проверка лицензий.



Содержание

1 Особенности DevOps для C++

2 Python как язык рецептов

3 Python как язык инфраструктуры

4 Устройство и развитие открытой разработки

Устройство и развитие открытой разработки

Можно публиковать свои пакеты!

Пакеты в разных пространствах user не конфликтуют.

На открытом bintray пространство имен резервировано за пользователем.

Формат имени пакета — `mylibrary/1.2.3@user/channel`.

Пакеты Conan Center — `mylibrary/1.2.3` (ранее - `mylibrary/1.2.3@conan/stable`).

Conan Center

- Курируемый репозитарий пакетов
- Предсобранные артифакты
- Стандартное пространство имен
- Ограниченное тестирование совместимости

- Больше схож с Maven Central
- Больше схож с python.libhunt.com
- Менее схож с pypi, rubygems, crates.io
- Пакеты в Debian
- 'Желтые страницы интернета' — `awesome-whatever`

Участие в открытой разработке

1. Свой пакет можно *связать* с Conan Center.

Участие в открытой разработке

1. Свой пакет можно *связать* с Conan Center.

2. Просто бинарник не примут.
   Нужен свой репозитарий на GitHub с исходным кодом и настроенным CI.

Участие в открытой разработке

1. Свой пакет можно *связать* с Conan Center.

2. Просто бинарник не примут.
   Нужен свой репозитарий на GitHub с исходным кодом и настроенным CI.
3. Сделать Pull Request в conan-io/conan-center-index .
   На каждый PR прогоняется CI и линтеры.

Участие в открытой разработке

1. Свой пакет можно *связать* с Conan Center.

2. Просто бинарник не примут.
   Нужен свой репозитарий на GitHub с исходным кодом и настроенным CI.
3. Сделать Pull Request в conan-io/conan-center-index .
   На каждый PR прогоняется CI и линтеры.

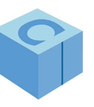
Способы (1) и (2) deprecated. PR — наше всё.

Сервера и репозитарии Conan

conan_server
Референсная открытая реализация на Bottle
Обычно запускается через WSGI (gunicorn)

Официальный репозитарий для открытых пакетов.
Публикация пакетов через связь собственного репозитария Bintray, либо через Conan Center Index.

Community Edition
Лучше всего подходит для приватного развертывания.



Проблемы и направления развития инфраструктуры

1. Наполнение Conan Center Index.
   Вишлист пакетов —
   https://github.com/conan-io/conan-center-index/issues/621
2. Упрощение рецептов в сторону декларативности.
3. Развитие инфраструктуры расширений.
   Ранее обнаружение проблем в PR
4. Проблемы комбинаторного взрыва конфигураций.
5. Python 3.6

Спасибо за внимание!

Присылайте ваши пакеты и пожелания и присоединяйтесь к разработке - https://github.com/conan-io

Всегда готов помочь и рассказать больше - `@theirix` .

Image attribution: Goodreads, Interstellar, JFrog

Version: 1.2

Спасибо за внимание!

Слайды доклада будут здесь на сайте конференции и в блоге:
https://omniverse.ru/blog/2020/09/11/rpw-pyconan

Присоединяйтесь к C++ User Group Moscow - канал
https://t.me/cppmoscow