链滴

xmake v2.2.5, 更加完善的 C/C++ 包依赖管理

作者：waruqi

原文链接：https://ld246.com/article/1554168528320

来源网站：链滴

此版本耗时四个多月，对包依赖管理进行了重构改进，官方仓库新增了mysql，ffmpeg等常用依赖包并且新增了大量新特性。

最近我打算对xmake的包仓库xmake-repo扩充一些常用的C/C++包。

大家有哪些经常使用到的包，都可以提到 https://github.com/xmake-io/xmake-repo/issues/10 面去，我之后会优先入库进去，提供给用户快速集成和使用依赖包。

关于新特性的详细说明见文章下文。

- 项目源码
- 官方文档
- 个人主页

第三方包管理器支持

新版本对内置的包管理进行了重构，已经支持的非常完善了，我们可以通过

add_requires("libuv master", "ffmpeg", "zlib 1.20.*")`

方便的安装使用依赖包，但是官方的包仓库xmake-repo目前收录的包还非常少，因此为了扩充xmak的包仓库，

xmake新增了对第三方包管理器的内置支持，通过包命名空间显式指定其他包管理器中的包，目前支对conan::，brew::和vcpkg::包管理中的包进行安装。

安装homebrew的依赖包

```
add_requires("brew::zlib", {alias = "zlib"}})
add_requires("brew::pcre2/libpcre2-8", {alias = "pcre2"}})

target("test")
    set_kind("binary")
    add_files("src/*.c")
    add_packages("pcre2", "zlib")
```

安装vcpkg的依赖包

```
add_requires("vcpkg::zlib", "vcpkg::pcre2")

target("test")
    set_kind("binary")
    add_files("src/*.c")
    add_packages("vcpkg::zlib", "vcpkg::pcre2")
```

不过需要注意的是，使用vcpkg，需要先对vcpkg与xmake进行集成才行，详细操作如下：

windows上用户装完vcpkg后，执行$ vcpkg integrate install，xmake就能自动从系统中检测到vcpg的根路径，然后自动适配里面包。

当然，我们也可以手动指定vcpkg的根路径来支持：

```
$ xmake f --vcpkg=f:\vcpkg
```

安装conan的依赖包

新版本实现了对conan的generator，来集成获取conan中的包信息，我们在xmake中使用也是非常方便，并且可以传递conan包的所有配置参数。

```
add_requires("conan::zlib/1.2.11@conan/stable", {alias = "zlib", debug = true})
add_requires("conan::OpenSSL/1.0.2n@conan/stable", {alias = "openssl", configs = {options = "OpenSSL:shared=True"}})

target("test")
    set_kind("binary")
    add_files("src/*.c")
    add_packages("openssl", "zlib")
```

执行xmake进行编译后：

```
ruki:test_package ruki$ xmake
checking for the architecture ... x86_64
checking for the Xcode directory ... /Applications/Xcode.app
checking for the SDK version of Xcode ... 10.14
note: try installing these packages (pass -y to skip confirm)?
  -> conan::zlib/1.2.11@conan/stable  (debug)
  -> conan::OpenSSL/1.0.2n@conan/stable
please input: y (y/n)

  => installing conan::zlib/1.2.11@conan/stable .. ok
  => installing conan::OpenSSL/1.0.2n@conan/stable .. ok

[  0%]: ccache compiling.release src/main.c
[100%]: linking.release test
```

内置依赖包查找支持

之前的版本提供了lib.detect.find_package来对依赖库进行查找，但是这需要通过import后才能使用并且一次只能查找一个包，比较繁琐：

```
target("test")
    set_kind("binary")
    add_files("src/*.c")
    on_load(function (target)
        import("lib.detect.find_package")
        target:add(find_package("openssl"))
        target:add(find_package("zlib"))
    end)
```

而新版本中通过内置find_packages接口，对lib.detect.find_package进行了进一步的封装，来提升用性：

```
target("test")
    set_kind("binary")
    add_files("src/*.c")
```

```
    on_load(function (target)
        target:add(find_packages("openssl", "zlib"))
    end)
```

并且还支持从指定的第三方包管理器中进行查找：

```
find_packages("conan::OpenSSL/1.0.2n@conan/stable", "brew::zlib")
```

参数配置依赖包安装

新版本中对内置的包管理进行了大规模重构和升级，并且对参数可配置编译安装依赖包进行了更好的持，我们可以在包仓库中定义一些编译安装配置参数，来定制安装包。

例如, 我们以pcre2的包为例：

```
package("pcre2")

    set_homepage("https://www.pcre.org/")
    set_description("A Perl Compatible Regular Expressions Library")

    set_urls("https://ftp.pcre.org/pub/pcre/pcre2-$(version).zip",
             "ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre/pcre2-$(version).zip")

    add_versions("10.23", "6301a525a8a7e63a5fac0c2fbfa0374d3eb133e511d886771e097e42707094a")
    add_versions("10.30", "3677ce17854fffa68fce6b66442858f48f0de1f537f18439e4bd2771f8bc7fb")
    add_versions("10.31", "b4b40695a5347a770407d492c1749e35ba3970ca03fe83eb2c35d4433a5a444")

    add_configs("shared", {description = "Enable shared library.", default = false, type = "boolen"})
    add_configs("jit", {description = "Enable jit.", default = true, type = "boolean"})
    add_configs("bitwidth", {description = "Set the code unit width.", default = "8", values = {"8", "16", "32"}})
```

上面我们通过add_configs定义了三个条件配置参数，使得用户在集成使用pcre2库的时候，可以定化选择是否需要启用jit版本、bit位宽版本等，例如：

```
add_requires("pcre2", {configs = {jit = true, bitwidth = 8}})
```

而且，配置参数是强约束检测的，如果传的值不对，会提示报错，避免传递无效的参数进来，像bitwidh参数配置，被限制了只能在values = {"8", "16", "32"}里面取值。

那么，用户如何知道我们的包当前支持哪些配置参数呢，很简单，我们可以通过下面的命令，快速查pcre2包的所有信息：

```
$ xmake require --info pcre2
```

输出结果如下：

```
The package info of project:
    require(pcre2):
```

```
    -> description: A Perl Compatible Regular Expressions Library
    -> version: 10.31
    -> urls:
      -> https://ftp.pcre.org/pub/pcre/pcre2-10.31.zip
        -> b4b40695a5347a770407d492c1749e35ba3970ca03fe83eb2c35d44343a5a444
      -> ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre/pcre2-10.31.zip
        -> b4b40695a5347a770407d492c1749e35ba3970ca03fe83eb2c35d44343a5a444
    -> repo: local-repo /Users/ruki/projects/personal/xmake-repo/
    -> cachedir: /Users/ruki/.xmake/cache/packages/p/pcre2/10.31
    -> installdir: /Users/ruki/.xmake/packages/p/pcre2/10.31/23b52ca1c6c8634f5f935903c9e
ea0e
    -> fetchinfo: 10.31, system, optional
      -> linkdirs: /usr/local/Cellar/pcre2/10.32/lib
      -> defines: PCRE2_CODE_UNIT_WIDTH=8
      -> links: pcre2-8
      -> version: 10.32
      -> includedirs: /usr/local/Cellar/pcre2/10.32/include
    -> platforms: linux, macosx
    -> requires:
      -> plat: macosx
      -> arch: x86_64
      -> configs:
        -> vs_runtime: MT
        -> shared: false
        -> jit: true
        -> bitwidth: 8
        -> debug: false
    -> configs:
      -> shared: Enable shared library. (default: false)
      -> jit: Enable jit. (default: true)
      -> bitwidth: Set the code unit width. (default: 8)
        -> values: {"8","16","32"}
    -> configs (builtin):
      -> debug: Enable debug symbols. (default: false)
      -> cflags: Set the C compiler flags.
      -> cxflags: Set the C/C++ compiler flags.
      -> cxxflags: Set the C++ compiler flags.
      -> asflags: Set the assembler flags.
      -> vs_runtime: Set vs compiler runtime. (default: MT)
        -> values: {"MT","MD"}
```

其中，里面的configs:部分，就是目前可提供配置的参数描述以及取值范围，而configs (builtin):中一些xmake内置的配置参数，用户也可直接配置使用。

例如最常用的debug模式包：

```
add_requires("pcre2", {configs = {debug = true}})
```

由于这个太过于常用，xmake提供了更方便的配置支持：

```
add_requires("pcre2", {debug = true})
```

另外requires:里面的内容，就是当前依赖包的配置状态，方便用户查看当前使用了哪个模式的包。

预处理模板配置文件

xmake提供了三个新的接口api，用于在编译前，添加一些需要预处理的配置文件，用于替代set_config_header等老接口。

- add_configfiles
- set_configdir
- set_configvar

其中add_configfiles相当于cmake中的configure_file接口，xmake中参考了它的api设计，并且在其础上进行了扩展支持，提供更多的灵活性。

此接口相比以前的set_config_header更加的通用，不仅用于处理config.h的自动生成和预处理，还以处理各种文件类型，而set_config_header仅用于处理头文件，并且不支持模板变量替换。

先来一个简单的例子：

```
target("test")
    set_kind("binary")
    add_files("src/*.c")
    set_configdir("$(buildir)/config")
    add_configfiles("src/config.h.in")
```

上面的设置，会在编译前，自动的将config.h.in这个头文件配置模板，经过预处理后，生成输出到指的build/config/config.h。

这个接口的一个最重要的特性就是，可以在预处理的时候，对里面的一些模板变量进行预处理替换，如：

config.h.in

```
#define VAR1 "${VAR1}"
#define VAR2 "${VAR2}"
#define HELLO "${HELLO}"

set_configvar("VAR1", "1")

target("test")
    set_kind("binary")
    add_files("main.c")

    set_configvar("VAR2", 2)
    add_configfiles("config.h.in", {variables = {hello = "xmake"}})
    add_configfiles("*.man", {copyonly = true})
```

通过set_configvar接口设置模板变量，裹着通过{variables = {xxx = ""}}中设置的变量进行替换处理。

预处理后的文件config.h内容为：

```
#define VAR1 "1"
#define VAR2 "2"
#define HELLO "xmake"
```

而{copyonly = true}设置，会强制将*.man作为普通文件处理，仅在预处理阶段copy文件，不进行

量替换。

默认的模板变量匹配模式为${var}，当然我们也可以设置其他的匹配模式，例如，改为@var@匹配规
：

```
target("test")
    add_configfiles("config.h.in", {pattern = "@(.-)@"})
```

我们也有提供了一些内置的变量，即使不通过此接口设置，也是可以进行默认变量替换的：

```
${VERSION} -> 1.6.3
${VERSION_MAJOR} -> 1
${VERSION_MINOR} -> 6
${VERSION_ALTER} -> 3
${VERSION_BUILD} -> set_version("1.6.3", {build = "%Y%m%d%H%M"}) -> 201902031421
${PLAT} and ${plat} -> MACOS and macosx
${ARCH} and ${arch} -> ARM and arm
${MODE} and ${mode} -> DEBUG/RELEASE and debug/release
${DEBUG} and ${debug} -> 1 or 0
${OS} and ${os} -> IOS or ios
```

例如：

config.h.in

```
#define CONFIG_VERSION "${VERSION}"
#define CONFIG_VERSION_MAJOR ${VERSION_MAJOR}
#define CONFIG_VERSION_MINOR ${VERSION_MINOR}
#define CONFIG_VERSION_ALTER ${VERSION_ALTER}
#define CONFIG_VERSION_BUILD ${VERSION_BUILD}
```

config.h

```
#define CONFIG_VERSION "1.6.3"
#define CONFIG_VERSION_MAJOR 1
#define CONFIG_VERSION_MINOR 6
#define CONFIG_VERSION_ALTER 3
#define CONFIG_VERSION_BUILD 201902031401
```

我们还可以对#define定义进行一些变量状态控制处理：

config.h.in

```
${define FOO_ENABLE}
```

```
set_configvar("FOO_ENABLE", 1) -- or pass true
set_configvar("FOO_STRING", "foo")
```

通过上面的变量设置后，${define xxx}就会替换成：

```
#define FOO_ENABLE 1
#define FOO_STRING "foo"
```

或者（设置为0禁用的时候）

```
/* #undef FOO_ENABLE */
/* #undef FOO_STRING */
```

这种方式，对于一些自动检测生成config.h非常有用，比如配合option来做自动检测：

```
option("foo")
    set_default(true)
    set_description("Enable Foo")
    set_configvar("FOO_ENABLE", 1) -- 或者传递true，启用FOO_ENABLE变量
    set_configvar("FOO_STRING", "foo")

target("test")
    add_configfiles("config.h.in")

    -- 如果启用foo选项 -> 天剑 FOO_ENABLE 和 FOO_STRING 定义
    add_options("foo")
```

config.h.in

```
${define FOO_ENABLE}
${define FOO_STRING}
```

config.h

```
#define FOO_ENABLE 1
#define FOO_STRING "foo"
```

关于option选项检测，以及config.h的自动生成，有一些辅助函数，可以看下：https://github.com/make-io/xmake/issues/342

除了#define，如果想要对其他非#define xxx也做状态切换处理，可以使用 ${default xxx 0} 模式，置默认值，例如：

```
HAVE_SSE2 equ ${default VAR_HAVE_SSE2 0}
```

通过set_configvar("HAVE_SSE2", 1)启用变量后，变为HAVE_SSE2 equ 1，如果没有设置变量，则用默认值：HAVE_SSE2 equ 0

关于这个的详细说明，见：https://github.com/xmake-io/xmake/issues/320

更加方便的特性检测

我们通过add_configfiles配合option检测，可以做到检测一些头文件、接口函数、类型、编译器特性否存在，如果存在则自动写入config.h中，例如:

```
option("foo")
    set_default(true)
    set_description("Has pthread library")
    add_cincludes("pthread.h")
    add_cfuncs("pthread_create")
    add_links("pthread")
    set_configvar("HAS_PTHREAD", 1)

target("test")
```

```
    add_configfiles("config.h.in")
    add_options("pthread")
```

config.h.in

```
${define HAS_PTHREAD}
```

config.h

```
#define HAS_PTHREAD 1
```

上面的配置，我们通过option检测pthread.h里面的接口以及link库是否都存在，如果能正常使用pthread库，那么自动在config.h中定义HAS_PTHREAD，并且test target中追加上相关的links。

上面的option可以支持各种检测，但是配置上少许复杂繁琐了些，为了让xmake.lua更加的简洁直观对于一些常用检测，xmake通过扩展includes接口，

提供了一些内置封装好的辅助接口函数，来快速实现上面的option检测，写入config.h的功能。

上面的代码我们可以简化为：

```
includes("check_cfuncs.lua")
target("test")
    add_configfiles("config.h.in")
    configvar_check_cfuncs("HAS_PTHREAD", "pthread_create", {includes = "pthread.h", links = "pthread"})
```

除了configvar_check_cfuncs，我们还有check_cfuncs函数，仅吧检测结果直接在编译时候追加，不写入configfiles文件中。

我们再来看个综合性的例子：

```
includes("check_links.lua")
includes("check_ctypes.lua")
includes("check_cfuncs.lua")
includes("check_features.lua")
includes("check_csnippets.lua")
includes("check_cincludes.lua")

target("test")
    set_kind("binary")
    add_files("*.c")
    add_configfiles("config.h.in")

    configvar_check_ctypes("HAS_WCHAR", "wchar_t")
    configvar_check_cincludes("HAS_STRING_H", "string.h")
    configvar_check_cincludes("HAS_STRING_AND_STDIO_H", {"string.h", "stdio.h"})
    configvar_check_ctypes("HAS_WCHAR_AND_FLOAT", {"wchar_t", "float"})
    configvar_check_links("HAS_PTHREAD", {"pthread", "m", "dl"})
    configvar_check_csnippets("HAS_STATIC_ASSERT", "_Static_assert(1, \"\");")
    configvar_check_cfuncs("HAS_SETJMP", "setjmp", {includes = {"signal.h", "setjmp.h"}})
    configvar_check_features("HAS_CONSTEXPR", "cxx_constexpr")
    configvar_check_features("HAS_CONSEXPR_AND_STATIC_ASSERT", {"cxx_constexpr", "c_static_assert"}, {languages = "c++11"})
```

config.h.in

```
${define HAS_STRING_H}
${define HAS_STRING_AND_STDIO_H}
${define HAS_WCHAR}
${define HAS_WCHAR_AND_FLOAT}
${define HAS_PTHREAD}
${define HAS_STATIC_ASSERT}
${define HAS_SETJMP}
${define HAS_CONSTEXPR}
${define HAS_CONSEXPR_AND_STATIC_ASSERT}
```

config.h

```
/* #undef HAS_STRING_H */
#define HAS_STRING_AND_STDIO_H 1
/* #undef HAS_WCHAR */
/* #undef HAS_WCHAR_AND_FLOAT */
#define HAS_PTHREAD 1
#define HAS_STATIC_ASSERT 1
#define HAS_SETJMP 1
/* #undef HAS_CONSTEXPR */
#define HAS_CONSEXPR_AND_STATIC_ASSERT 1
```

可以看到，xmake还提供了其他的辅助函数，用于检测：c/c++类型，c/c++代码片段，c/c++函数口，链接库，头文件是否存在，甚至是c/c++编译器特性支持力度等。

关于这块的更加完整的说明，可以看下：https://github.com/xmake-io/xmake/issues/342

配置自定义安装文件

对于xmake install/uninstall命令，xmake新增了add_installfiles接口来设置一些安装文件，比起on_install，此接口用起来更加的方便简洁，基本能够满足大部分安装需求。

比如我们可以指定安装各种类型的文件到安装目录：

```
target("test")
    add_installfiles("src/*.h")
    add_installfiles("doc/*.md")
```

默认在linux等系统上，我们会安装到/usr/local/*.h, /usr/local/*.md，不过我们也可以指定安装到特子目录：

```
target("test")
    add_installfiles("src/*.h", {prefixdir = "include"})
    add_installfiles("doc/*.md", {prefixdir = "share/doc"})
```

上面的设置，我们会安装到/usr/local/include/*.h, /usr/local/share/doc/*.md

我们也可以通过()去提取源文件中的子目录来安装，例如：

```
target("test")
    add_installfiles("src/(tbox/*.h)", {prefixdir = "include"})
    add_installfiles("doc/(tbox/*.md)", {prefixdir = "share/doc"})
```

我们把src/tbox/*.h中的文件，提取tbox/*.h子目录结构后，在进行安装：/usr/local/include/tbox/*.

, /usr/local/share/doc/tbox/*.md

当然，用户也可以通过set_installdir接口，来配合使用。

关于此接口的详细说明，见：https://github.com/xmake-io/xmake/issues/318

CMakelists.txt导出

新版本对xmake project工程生成插件进行了扩展，新增了对CMakelists.txt文件的导出支持，方便用xmake的用户可以快速导出CMakelists.txt提供给cmake，

以及CLion等一些支持cmake的工具使用，使用方式如下：

```
$ xmake project -k cmakelists
```

即可在当前工程目录下，生成对应的CMakelists.txt文件。

更新内容

新特性

- 添加 string.serialize和string.deserialize去序列化，反序列化对象，函数以及其他类型
- 添加 xmake g --menu去图形化配置全局选项
- #283: 添加target:installdir()和set_installdir()接口
- #260: 添加add_platformdirs接口，用户现在可以自定义扩展编译平台
- #310: 新增主题设置支持，用户可随意切换和扩展主题样式
- #318: 添加add_installfiles接口到target去自定义安装文件
- #339: 改进add_requires和find_package使其支持对第三方包管理的集成支持
- #327: 实现对conan包管理的集成支持
- 添加内置API find_packages("pcre2", "zlib")去同时查找多个依赖包，不需要通过import导入即直接调用
- #320: 添加模板配置文件相关接口，add_configfiles和set_configvar
- #179: 扩展xmake project插件，新增CMakelist.txt生成支持
- #361: 增加对vs2019 preview的支持
- #368: 支持private, public, interface属性设置去继承target配置
- #284: 通过add_configs()添加和传递用户自定义配置到package()
- #319: 添加add_headerfiles接口去改进头文件的设置
- #342: 为includes()添加一些内置的辅助函数，例如：check_cfuncs

改进

- 针对远程依赖包，改进版本和调试模式切换
- #264: 支持在windows上更新dev/master版本，xmake update dev
- #293: 添加xmake f/g --mingw=xxx 配置选线，并且改进find_mingw检测

- #301: 改进编译预处理头文件以及依赖头文件生成，编译速度提升30%
- #322: 添加option.add_features, option.add_cxxsnippets 和 option.add_csnippets
- 移除xmake 1.x的一些废弃接口, 例如： add_option_xxx
- #327: 改进lib.detect.find_package增加对conan包管理器的支持
- 改进 lib.detect.find_package并且添加内建的find_packages("zlib 1.x", "openssl", {xxx = ...})接口
- 标记 set_modes()作为废弃接口， 我们使用add_rules("mode.debug", "mode.release")来替代它
- #353: 改进target:set, target:add 并且添加target:del去动态修改target配置
- #356: 添加qt_add_static_plugins()接口去支持静态Qt sdk
- #351: 生成vs201x插件增加对yasm的支持
- 重构改进整个远程依赖包管理器，更加快速、稳定、可靠，并提供更多的常用包

Bugs修复

- 修复无法通过 set_optimize() 设置优化选项，如果存在add_rules("mode.release")的情况下
- #289: 修复在windows下解压gzip文件失败
- #296: 修复option.add_includedirs对cuda编译不生效
- #321: 修复PATH环境改动后查找工具不对问题

原文：https://tboox.org/cn/2019/03/29/xmake-update-v2.2.5/