WebAssembly 探索与实践

黄烈锦 金山办公 Kingsoft Office

WebAssembly 在探索与实践中的一些问题

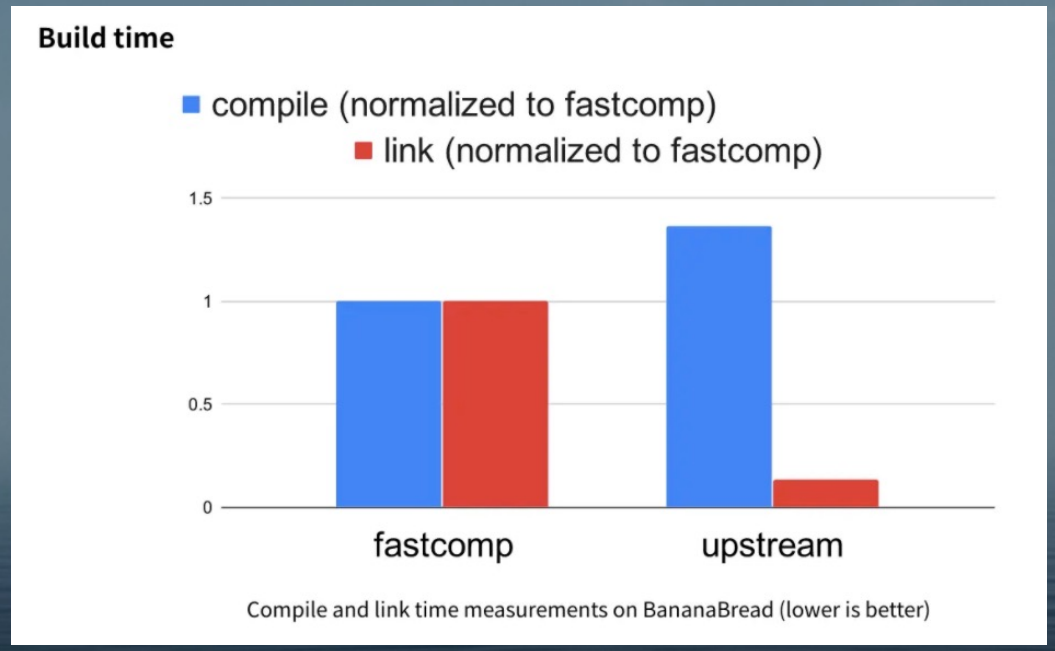
工具链 → 类型 → IO → 缓存 → 性能 → 安全

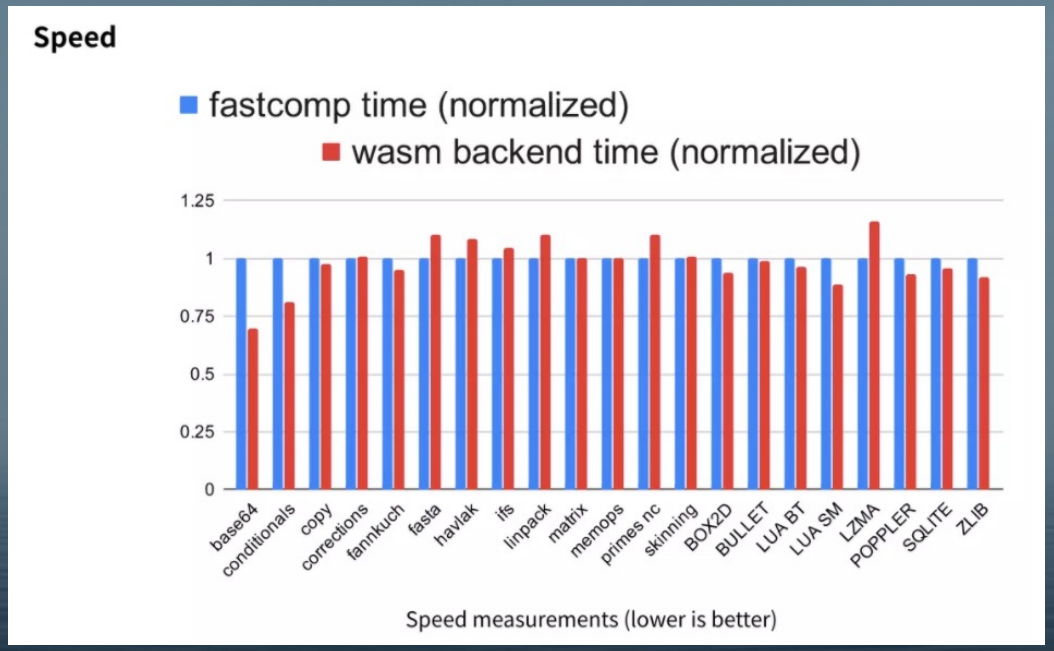
工具链

Emscripten和编译器前后端支持

旧版本：fastcomp

新版本：llvm-webassembly





Clang vs GCC vs Emcc

结合CMake

Emscripten.cmake

```
# Clean
    cd build.emscripten;
    if [ -f "cpplib.wasm" ]
    then
        rm cpplib.*;
    else
        echo "[CLEAN] Empty wasm file.";
    fi
    cd ../build;
    rm -rf *;

# Build
    cmake -DCMAKE_TOOLCHAIN_FILE=~/emsdk/fastcomp/emscripten/cmake/Modules/Platform/Emscripten.cmake ..;
    emmake make VERBOSE=1;
```

set_target_properties

```
# 对生成库进行链接
link_directories("build/lib")
link_libraries("cpplib")

# 收集头文件和源文件到相应变量
file(GLOB_RECURSE CORE_HDR src/*.h)
file(GLOB_RECURSE CORE_SRC src/*.cpp)

# 设置cpp版本
add_definitions("-std=c++17")
add_executable(cpplib ${CORE_SRC} ${CORE_HDR})

# Emcc / Em++ 编译参数
set_target_properties(cpplib PROPERTIES LINK_FLAGS "-s FULL_ES3=1 \
                                                    -Oz \
                                                    -s WASM=1 \
                                                    -s ERROR_ON_UNDEFINED_SYMBOLS=1 \
                                                    -s DEMANGLE_SUPPORT=1 \
                                                    --bind \
                                                    --pre-js ${CMAKE_SOURCE_DIR}/.../pre.js \
                                                    --js-library ${CMAKE_SOURCE_DIR}/.../js-lib.js
                                                    ")

```

类型与IO

1. 基本的共识
   - 类型约定
   - WebIDL
2. 可能出现的问题
   - 存在从 wasm lang --- js ---- wasm lang 这样的调度
   - 进行细致的指针与内存的管理
   - 存在语言特性导致开发成本增加的情况
     例如：当需要从虚拟FS中进行加载资源时，可能碰到编解码，IO等关键问题，但某些语言在某些类型支持上API不那么完整，需要引入其他支持库 或者 通过JS来实现
3. 如何解决

WASM加载加速

1. 方式
   - 通用的Web多级缓存方式
   - 考虑进行本地的WASM文件持久化
2. 实践落地的碰到的问题
   - 大型库的wasm文件过大
   - 本地持久化在转码上耗费资源，而且base64格式并非最理想的格式
3. 思考
   - 通过combo优化方式实现通用可降级加载工具，虽然可能入侵业务代码，但是在一定程度上拆分业务库的方式进行细粒度优化加载

性能

1. 我们发现的问题
   - WASM提高计算能力上提升幅度巨大
   - 在实际集成中，
     - 当我们进行诸多弥补WASM缺陷的机制（IO，编译优化与Worker中ES6支持等）需要进行一定程度上的性能，内存，可维护性之间进行不断benchmark抉择
     - 以下为10000次测试数据取均值得出的wasm包情况：

优化途径	task.js	task.wasm	编译	压缩GlueJS	影响执行	执行效率
原始	329k	720k	√	-	-	-
清除无关代码	304k	707k	√	-	-	1.01~1.20ms
emcc编译器-O1	234k	274k	√	√	-	0.90~1.10ms
emcc编译器-O2	129k	291k	ES6编译失败	√	-	0.8~1.0ms
emcc编译器-O3	125k	289k	ES6编译失败	√	-	0.8~0.9ms
emcc编译器-Os	125k	262k	ES6编译失败	√	-	0.8~0.9ms
emcc编译器-Oz	125k	260k	ES6编译失败	√	-	0.8~0.9ms
GCC谷歌编译-closure 1	-	-	-	只在-O1优化级别下生效	√	-

安全

有沙箱就安全了吗

造成安全问题的几种情况

几种情况：
Indirect function calls -> XSS
Buffer Over ow -> XXS

emscripten_run_script()

int emscripten_run_script_int(const char *script)
char emscripten_run_script_string(const char script)
void emscripten_async_run_script(const char *script, int millis)
void emscripten_async_load_script(const char *script, em_callback_func onload, em_callback_func onerror)

如何进行安全编码？

- 遵循最佳C/C++编程规则: 开发人员应该意识到WASM仍处于开发的最初阶段，并且在未来几年内可能会发现更多问题。为本地编译建立的所有规则都是相关的，并且在编译为WebAssembly时应遵循这些规则。在WASM中和在本机代码中一样严肃对待C语言安全问题。

- 避免emscripten_run_script: 从WASM中动态执行JavaScript是一种危险的模式。如果存在类型混淆或溢出到函数指针等问题，那么这些函数的存在将允许漏洞利用代码直接执行JavaScript

- 使用Clang的CFI编译时，使用Clang的Control Integrity flag(-fsanitize = c)可以防止某些函数指针操作问题

- 使用优化可以删除一些可以用于涉及函数指针操作的漏洞的编译器的构建功能。

Q & A

KingSoft Office 黄烈锦

WeChat: Copyleft4U, 一起交流

THANK YOU