[翻译]从C++编译WebAssembly的实用指南

小蘑菇糖糖
web三脚猫 / 萌新摄影 / Ingress Res
+ 关注他

大概在7月份的时候，我重构了公司直播项目中的语音录制模块，基本收敛了线上已有的问题，并大幅减少了录音文件编码时间。核心部分是opus-recorder，这个项目最新版本使用WebAssembly重构了录音文件的编码过程。效果很好，5min以上的录音基本上从之前的20s+的编码时间缩短到了5s左右。也因此对WebAssembly产生了兴趣，花了一些时间去看文章和写demo尝试。
以下是一篇不错的入门文章，英文内容，这里翻译一下。初次翻译，如有疏漏或错误之处，还望大家指正。当然在这个时间节点WebAssembly并不是什么新鲜玩意儿，网上也有很多教程了，仅效抛砖引玉，也希望在一些地方可以提供帮助。

从C++编译至WebAssembly的实用指南

大多数我认识的C程序员都听说过WebAssembly，但很多人在开始阶段就遇到了麻烦。本指南将为您带来一个简单的“Hello World”实例，一个在C和JavaScript之间具有交互能力的状态应用程序。在超出最小限度之外我没找到一篇单独的文章。实际上从最简单的“Hello World”到系统，需要花费很多精力才能解决现实中的实际问题。本文目的即是这个。
对了，最终的代码放在这个仓库（github.com/tom-010/weba...），但是遵循本教程更加有意义。
注意：本文没有讲到如何把数组从JS传到WebAssembly中，相关内容在这篇文章
本文并不介绍WebAssembly本身或者讨论为什么你该使用它，所以开头并没有大段讲演的动机。
虽然如此这里还是列出WebAssembly官方的定义：

> WebAssembly（缩写为Wasm）是基于堆栈的虚拟机的二进制指令格式。 Wasm被设计为可移植目标，用于编译高级语言（如C / C ++ / Rust），从而可以在Web上为客户端和服务器应用程序进行部署。

然而，老实讲，唯一的理由就是第一点：高效和速度。其他任何功能JS实现都更好一些。

那么，让我们开始吧！

关于操作系统

我使用的是Ubuntu 18.10，以及C++的标准构建工具:

```
$ apt install build-essential cmake python git
```

这些工具在 Windows 和MacOS上同样也有良好的支持。

编译工具链

首先，我们需要工具链（基于Clang）。最好的起点是这篇文章： Getting Started Guide（这里也有其他操作系统的步骤）。

```
$ mkdir ~/tmp && cd ~/tmp
git clone https://github.com/juj/emsdk.git
cd emsdk
./emsdk install --build=Release sdk-incoming-64bit binaryen-master-64bit
./emsdk activate --build=Release sdk-incoming-64bit binaryen-master-64bit
```

这会花点时间（在其他Clang编译好之前），同时也需要一些磁盘空间。
可以随意花点时间来"WebAssembly Explorer"上体验一把WebAssembly（类似编译器的浏览工具）。

首次编译："Hello World"

现在来编译"Hello World"吧：

```
$ cat hello.cpp

#include<iostream>
int main() {
  std::cout << "Hello World" << std::endl;
  return 0;
}
```

在编译前，我们先要初始化已经编译好的工具链。在你之前clone的目录下执行下面这条命令：

```
$ source ./emsdk_env.sh --build=Release
```

更好的做法是在你'.bashrc'文件追加下面的这行代码（同样可以在clone好的文件夹下面运行通过）：

```
$ echo "source $(pwd)/emsdk_env.sh --build=Release > /dev/null" >> ~/.bashrc
```

编译

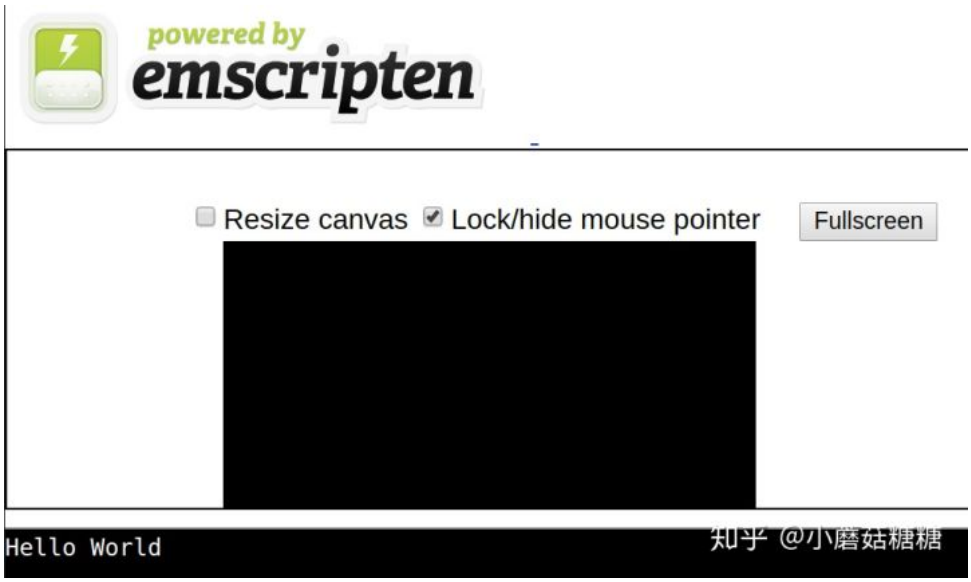
```
$ em++ hello.cpp -s WASM=1 -o hello.html
```

运行代码

如果你在浏览器直接打开index.html的话，会遇到CORS-Problems问题。你必须通过Web服务器为其提供服务。
EmScripten自带了这个服务：

```
$ emrun --port 8080
```

上面的命令会启动一个Web服务，打开浏览器并导航至当前目录。只需单击新创建的hello.html，瞧：



执行结果

Emscripten 提供了一个控制台来执行你的代码。

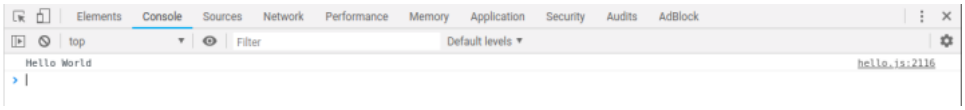
自行调用（JavaScript）

浏览器中的控制台很nice但是在生产环境中不那么好用。来写个最小化脚本来调用我们的“Hello World”。
Emscriptes同时也生成了一个hello.html和一个hello.js文件。HTML文件过分臃肿，其中没有任何对于进一步使用特别有用的文件。hello.js文件则非常有帮助，它加载并实例化我们的WebAssembly代码，并为其提供JavaScript接口。因此，我们保留它并用以下内容替换HTML文件：

```
$ cat index.html

<html>
  <body>
    <script src="hello.js"></script>
  </body>
</html>
```

刷新浏览器，打开开发者工具中console面板，然后可以看到我们得到：



“Hello World” in the web-console

两个或更多的文件

来个一次性代码（斐波拉切数字）：

```
$ cat fib.cpp

int fib(int x) {
  if (x < 1)
    return 0;
  if (x == 1)
    return 1;
  return fib(x-1)+fib(x-2);
}

$ cat hello.cpp

#include<iostream>
#include "fib.cpp"
int main() {
  std::cout << "fib(5) = " << fib(5) << std::endl;
  return 0;
}
```

执行以下命令进行编译：

```
$ em++ hello.cpp -s WASM=1
```

我的Web服务还在跑着，所以刷新后显示如下：

fib(5) = 5

好消息是：它跑起来了。坏消息是：我的hello.html文件被重写了。解决方法是指定输出'hello.js'来取代'hello.html'：

```
$ em++ hello.cpp -s WASM=1 -o hello.js
```

重新输出的文件就只有'hello.wasm' 和 'hello.js'文件，没有'hello.html'。来一点自动化工作，一个构建脚本和一个适当的文件夹结构：

```
$ cat build.shrm build/ -rfmkdir build
$ cd build
$ em++ ../cpp/hello.cpp -s WASM=1 -o hello.js
$ mv hello.js ../web/gen/
$ mv hello.wasm ../web/gen/
```

```
$ tree ..
├── build
├── build.sh
├── cpp
│   ├── fib.cpp
│   └── hello.cpp
├── serve.sh
└── web
    ├── gen
    │   ├── hello.js
    │   └── hello.wasm
    └── index.html
```

再来一个小小的方便输入的运行脚本:

```
$ cat serve.sh
emrun --port 8080 web/
```

一个调整过的index.html文件:

```
$ cat web/index.html
<html>
  <body>
    <script src="gen/hello.js"></script>
  </body>
</html>
```

很棒。现在我们可以独立开发Web应用，并将生成的源代码放在一个额外的文件夹中，这样就很容易记住，你不应该修改它们（就像任何生成的源文件一样）。

头文件

声称前面的例子有多文件是作弊的（没有头文件，等等）。所以我们把fib代码分拆成头文件和实现:

```
$ cat fib.h

#ifndef FIB
#define FIBint fib(int x);#endif$ cat fib.cpp#include "fib.h"int fib(int x) {
  if (x < 1)
    return 0;
  if (x == 1)
    return 1;
  return fib(x-1)+fib(x-2);
}

$cat hello.cpp

#include<iostream>
#include "fib.h"
int main() {
  std::cout << "fib(6) = " << fib(6) << std::endl;
  return 0;
}
```

注意hello.cpp不引用fib.cpp文件，而仅仅是头文件。因此必须进行链接过程，这就是为什么编译失败的原因了:

```
error: undefined symbol: _Z3fibi
warning: To disable errors for undefined symbols use `-s ERROR_ON_UNDEFINED_SYM
Error: Aborting compilation due to previous errors
shared:ERROR: '/home/thomas/tmp/emsdk/node/8.9.1_64bit/bin/node
/home/thomas/tmp/emsdk/emscripten/incoming/src/compiler.js
/tmp/tmpDR9qjf.txt
/home/thomas/tmp/emsdk/emscripten/incoming/src/library_pthread_stub.js' failed
```

把N加进编译脚本中可以修复这个问题：

```
$ cat build.sh
rm build/ -rf
mkdir build
cd build
em++ ../cpp/hello.cpp ../cpp/fib.cpp -s WASM=1 -o hello.js || exit 1
mv hello.js ../web/gen/
mv hello.wasm ../web/gen/
```

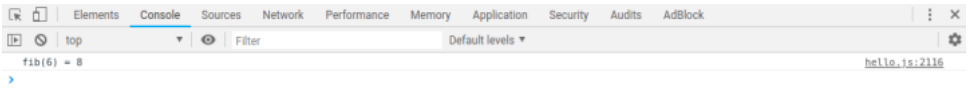
注意：*会让脚本在编译失败后终止！*

```
$ ./build.sh
$ ./serve.sh
```

构建任务现在通过了。请注意，我把N函数的参数改成了：

```
int main() {
 std::cout << "fib(6) = " << fib(6) << std::endl;
 return 0;
}
```

因此我们现在可以看到实际的差异了：



成功编译了多个文件！只要你能扩展那个简易构建脚本，这种方式就行得通。稍后我们将使用构建系统（b æX）来处理更复杂的项目。但在此之前先让我们来看下参数传递的问题吧！

反汇编

有时候把你的代码反汇编成g表达式是很有用的（比如下一节）。你可以通过下面这条命令来实现：

```
$ wasm-dis hello.wasm -o hello.wast
```

在这个文件（里，你可以轻松找到全局函数等。g表达式是的文本表示形式。要了解构建
中的模块，请查看的出色指南。
当b代码使用标记编译时反汇编会变得特别有用。然后，结果只有几行，您可以仔细分析它。

作为构建的一部分

```
$ cat build.shrm build/ -rf
mkdir build
cd build
em++ ../cpp/hello.cpp ../cpp/fib.cpp -g -s WASM=1 -o hello.js || exit 1
mv hello.js ../web/gen/
mv hello.wasm ../web/gen/
```

现在你可以在构建文件夹（6ő鿔ıŬX鿍ŋ̃Ţ ķm̀）中找到始终最新的ŋŢ ķmŋ文件。

函数调用和传递参数

控制台所展示的ŋN̈ɒн ɪ ʼl ⁾ ŋ来自b ̈l ̈l程序主函数中的数字，会在加载完后执行。现在我想在÷ ţ ģ\ļHạ̈нн中调用Nɒ̈J函数：

```
$ cat index.html
<html>
  <body>
    <script src="gen/hello.js"></script>
    <script>
     console.log( fib(10) );
    </script>
  </body>
</html>
```

现在，我面临两个问题：
⁵ 、当我想执行Nɒ̈J（⁵⁰ ）时，程序尚未加载
⁶ 、XXȟ ɭ没有从b ̈l ̈l中导出函数Nɒ̈J，因此在÷ģ不可用

从b ̈l ̈l中导出函数

注意：要获取所有导出的函数，可以通过ᵞ ŗ ‛ǢÊ ⁼ ǍĠdƷ@̣̄ŗ ‛ǢÊ ⁼ü dƷ@̣̄ŗ ‛Ǵᵥ反编译ŗ ‛ǢÊ文件，并在ŗ ‛Ǵ文件中搜索ᵞ ̀Ʒṡd̨üȅ́ᵥ。由于Q 函数名带有前缀。该文件用Ď̄ƷṡḍȅƷǦǦŇî写成。
U̇
如果不进行修改，则仅导出ŋ̈ȟ ạ̈ɳ函数。我们必须更改构建脚本：

```
$ cat build.shrm build/ -rf
mkdir build
cd build
em++ ../cpp/hello.cpp ../cpp/fib.cpp -s WASM=1 -s EXPORT_ALL=1 -o hello.js || e
mv hello.js ../web/gen/
mv hello.wasm ../web/gen/
```

使用ŋ̈ǒķ t ṣ Ē̇ ĖĶŨ2‡‡ ʼl ⁵ŋ选项就可以导出Nɒ̈J函数，但函数名会被b ̈l ̈l重命名为ŋ̈ǘ ⁷ Nɒ̈aŋ。我通过查看反编译的代码找到了这个函数名（不用担心，这很容易）。

控制编译的内容

如你所见，编译后的内容有好几个Ÿ^大（在我的例子里是ⁿ ŋ Ÿ^）。仅仅是一个非常简单的算法。大部分内容来自ȁ ķmK ȟ ̈模块。当你删除这个引用和调用，并在6ő鿔ŋ̈kŬ中设置标志时，只有我们的代码应出现在生成的Ţ ķȟ 文件中，该编译文件变得更加简便：

```
$ cat build.shrm build/ -rf
mkdir build
cd build
```

```
mv hello.wasm ../web/gen/
$ cat hell.cpp
// #include<iostream>
#include "fib.h"int main() {
  fib(10);
  return 0;
}
```

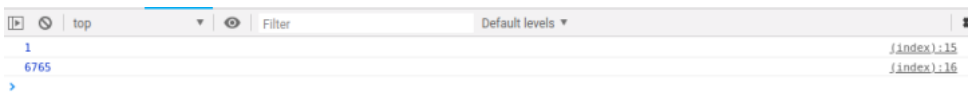
现在文件就只有[illegible]大小了。这样就比较合适了。然后我们来调用[illegible]函数：

```
$ cat hello.html<html>
<body>
<script>function loadWasm(fileName) {
  return fetch(fileName)
    .then(response => response.arrayBuffer())
    .then(bits => WebAssembly.compile(bits))
    .then(module => { return new WebAssembly.Instance(module) });
};

loadWasm('gen/hello.wasm')
  .then(instance => {
    let fib = instance.exports.__Z3fibi;
    console.log(fib(1));
    console.log(fib(20));
  });

</script>
</body>
</html>
```

注意，这里我们不再引用[illegible]。我们自己做了最小化的实现来加载[illegible]文件。在第一个代码块里，我们加载，编译和初始化了程序，在第二段里则执行了它。并不复杂。在本文的后面，我将处理一些内存方面的问题，但现在可以运行了。



[illegible]这是通过[illegible]调用的第一份[illegible]代码。

加载超过[illegible]的文件

想象一下，我们的代码越来越大。我通过重新引入[illegible]和在构建脚本中移除相应的[illegible]来模拟这个情况：

```
$ cat build.sh...
$ em++ ../cpp/hello.cpp ../cpp/fib.cpp -s WASM=1 -s EXPORT_ALL=1 -O3  ...
$ cat hello.cpp#include<iostream>
#include "fib.h"int main() {
  std::cout << fib(10) << std::endl;
  return 0;
}
```

[illegible]文件回到了[illegible]大小。这对于模拟部分情况下的代码足够真实了。我们没有更改绑定或算法的任何内容，所以代码应该可以正常运行：

错误发生了：

```
RangeError: WebAssembly.Instance is disallowed on the main thread, if the buffe
```

这是有道理的。我们不想通过加载，编译和加载Ŷ X62ķķXň 6ŭ文件来阻塞主线程。
关于如何高效地处理Ŷ X62ķķXň 6ŭ，õ”” nĵX提供了 优秀的文档。
实际上，从现在开始它变得非常讨厌，因为我们将为每个导出的函数定义内容。
否则，我们会收到许多有线错误。可以为某些功能定义所有绑定，但是请记住，我们导出了包括所有ŋ ă ķŋX ň ŋ的所有功能。我尝试了几个小时，才意识到t ň ģ\ļ ĀĦXę生成的代码是目前最简单的方法。因此：

```
$ cat index.html
<html>
<body>
<script src="gen/hello.js"></script>
<script>
Module.onRuntimeInitialized = function() {
  console.log(Module.__Z3fibi(30));
}
</script>
</body>
</html>
```

我仍然使用Ňŭ函数的前缀名称。我注册了函数ŋ Î ” l őĔXŋ ęĖőęm̃ň XĄęāă ĎẀXl ŋ，该函数可以确保在加载，编译和实例化（大型）程序之后执行该函数。起作用了：

虽然不是最好的解决方案，但它可以工作。Ŷ X62ķķXň 6ŭ仍然非常脆弱（至少在工具方面），因此我们必须忍受这一点。第一步是将导出的功能列入白名单。

有状态的b ˝l˝l代码

仅在极少数情况下，调用无状态函数才有意义。因此，我想设计一个简单的类：

```
$ cat fib.h#ifndef FIB
#define FIBclass Fib {
public:
  Fib();
  int next();private:
  int curr = 1;
  int prev = 1;
};
#endif
$ cat fib.cpp
#include "fib.h"
Fib::Fib() {}
int Fib::next() {
  int next = curr + prev;
  prev = curr;
  curr = next;
  return next;
}
$ cat hello.cpp
#include "fib.h"
#include <iostream>
int main() {
  Fib fib{};
  std::cout << fib.r
```

```
  std::cout << fib.next() << std::endl;
  std::cout << fib.next() << std::endl;
  return 0;
}
$ g++ hello.cpp fib.cpp -o fib && ./fib2
3
5
8
13
```

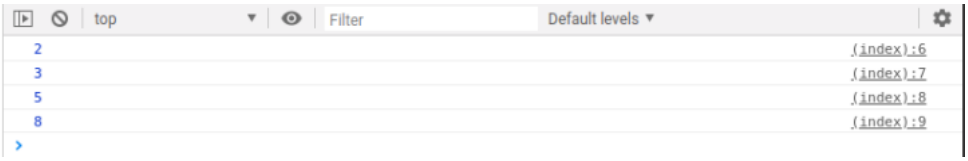
这里没什么特别的。只是一个转储和有状态的类。让我们通过÷ t gl¸ā使用它。我从小处着手，并在b "l"l中进行实例化：

```
$ cat hello.cpp
#include "fib.h"
int fib() {
  static Fib fib = Fib();
  return fib.next();
}
int main() {
  fib();
  return 0;
}
```

注意：我在Ê ‘Ã中使用了ìÀ调用，因为编译器不会优化我的ìÀ函数。
我的ńÑ绑定的名称已更改，因此我的÷g代码如下所示：

```
$ cat index.html
<html>
<body>
<script src="gen/hello.js"></script>
<script>
Module.onRuntimeInitialized = function() {
  console.log(Module.__Z3fibv());
  console.log(Module.__Z3fibv());
  console.log(Module.__Z3fibv());
  console.log(Module.__Z3fibv());
}
</script>
</body>
</html>
```

开箱即用：



通过调度的多个对象ı状态

每个公有函数一个对象（状态）是不够的。下一个最简单的方法是进行调度：

```
$ cat hello.cpp
#include "fib.h"
#include <vector>
auto instances = st
int next_val(int fi
```

```
int new_fib() {
  instances.push_back(Fib());
  return instances.size() - 1;
}
int main() {
  int fib1 = new_fib();
  next_val(fib1);
  return 0;
}
```

这个想法很简单。我从函数式编程中学到了它。ņȩXȚ ừǞǹ是我们的构造函数，而整数是其ņ地址ņ。也许不是最优雅的解决方案，但是它有效且易于理解，因此可以更改。对于所需的功能，我们有两个名称：

```
__Z7new_fibv
__Z8next_vali
```

调用很简单：

```
$ cat index.html
<html>
<body>
<script src="gen/hello.js"></script>
<script>
Module.onRuntimeInitialized = function() {
  let fib1 = Module.__Z7new_fibv();
  let fib2 = Module.__Z7new_fibv();  console.log(Module.__Z8next_vali(fib1));
  console.log(Module.__Z8next_vali(fib1));
  console.log(Module.__Z8next_vali(fib1));  console.log(Module.__Z8next_vali(f:
  console.log(Module.__Z8next_vali(fib2));
  console.log(Module.__Z8next_vali(fib2));
}
</script>
</body>
</html>
```

封装b˜l˜l：构建一个门面（ô ˯ I X)

> 译者注：门面模式（ô ˯ I X）又称外观模式，用于为子系统中的一组接口提供一个一致的界面。对这个概念不理解？可参考外观模式 或自行õ””nj̈X。

是时候抽象这些丑陋的b ˜l˜l接口了：

```
$ cat index.html
<html>
<body>
<script src="gen/hello.js"></script>
<script>
class Fib {
  constructor() {
```

```
  next() {
   return Module.__Z8next_vali(this.cppInstance);
  }
}
Module.onRuntimeInitialized = function() {
  let fib1 = new Fib();
  let fib2 = new Fib();  console.log(fib1.next());
  console.log(fib1.next());
  console.log(fib1.next());  console.log(fib2.next());
  console.log(fib2.next());
  console.log(fib2.next());
}
</script>
</body>
</html>
```

输出仍然相同，但是现在我们有了一个非常漂亮的÷ g接口并封装了b ˜l˜l部分：

更多b ˜l˜l对象的实例化

当然，下一步可能是在÷ g中实际调用ŋôǎ ŋ类的构造函数。但是，对我来说这很有意义（现在）。ẹXṮ ǔǹ ˜ĩ也是为÷ g优化的专业构造函数，并且是与语言无关的。用b替换我们的方法在实例化中不需要任何概念上的改变。我下一步是用Î Ḥ取代向量，并提供删除方法以摆脱不再需要的对象。

b ˜l˜l和÷ ṭ g⌉ḷ ȁḥn之间的稳定接口

如您所知，函数名在每次重构后都会更改，这导致我们的集成失败。这些奇怪的函数名来自于b ˜l˜l的名字修饰ḥẹ ȟ Xȟ ẹnḭẹnḭ。

一致的函数名

为防止这种情况，我们将其签名导出为b代码：

```
$ cat hello.cpp
#include "fib.h"
#include <vector>
extern "C" {
 int new_fib();
 int next_val(int fib_instance);
}
auto instances = std::vector<Fib>();
int next_val(int fib_instance) {
 return instances[fib_instance].next();
}
int new_fib() {
 instances.push_back(Fib());
 return instances.size() - 1;
}
int main() {
 int fib1 = new_fib
```

```
}
```

这很好，因为我本来就想列出导出的函数。现在，我们有了更一致的函数名（只是带有下划线的前缀）：

```
$ cat hello.cpp
<html>
<body>
<script src="gen/hello.js"></script>
<script>
class Fib {
  constructor() {
    this.cppInstance = Module._new_fib();
  }  next() {
    return Module._next_val(this.cppInstance);
  }
}
Module.onRuntimeInitialized = function() {
  // ...
}
</script>
</body>
</html>
```

仅导出实际使用的函数

你可能会认识到ἠķ⌊Ḩṁk̈ἠ的大小以及反编译的ἠ̤Ţ ķm̈ἠ文件中大量导出的函数，以及÷ģ上下文中ἠ̂ ”I őЁXἠ的结果大小。所有的这些都来自'Eș Ē' ĖĶǘ2‡‡'E

```
$ cat build.sh
$ rm build/ -rf
$ mkdir build
$ cd build
$ em++ ../cpp/hello.cpp ../cpp/fib.cpp -s WASM=1 -s EXPORT_ALL=1 -o hello.js |
$ mv hello.js ../web/gen/
$ mv hello.wasm ../web/gen/
```

t ȟ ģ⌊Ḩḿχę会导出所有引入软件包的所有函数，并为其生成绑定。使用一致的函数名，我们可以仅导出所需的内容。

```
$ cat build.sh
rm build/ -rf
mkdir build
cd build
em++ ../cpp/hello.cpp ../cpp/fib.cpp -s WASM=1 -s EXPORTED_FUNCTIONS="[_new_fil
mv hello.js ../web/gen/
mv hello.wasm ../web/gen/
```

Ŷ X⌊ ę ķḤX⌊ãŭŢ ãŭ 'Eș Ē' ĖĶt k ủóń Ôb ĶÆÔἠĘ Ǖ mŢ X Ţ ęṁṁ ķḤX⌊ãŭἠĶǕX nχęXḤ mXI 'ЁXЁ ṁķ" Ek
ę" Ţ ȟ ő⌊Ǖķȟ ЁXḩ ęI ŢXI " ę'ṁЁX áṁḩI ónǕäęṁXḩę Ek ęüȟ " ḩX ң⌊ǕX⌊ áäṁ" őṁἠ

我们可以使用ἠt ș Ē' ĖĶt k ủóń Ôb ĶÆÔἠ进行指定。现在，生成的ἠǕXЁ ṁķἠ要小得多，而且我们也不会再泄漏内部实现了（可以检查看看）。

集成

集成测试

请记住，如果逻辑上有缺陷，那么仅当两个组件本身的集成不再起作用时，集成测试才应该中断。本文不是有关÷gŏnXkmęnpŇHȟ XŢ" ļak的教程。因此，我只是编写没有测试运行程序等的普通÷ ţ gl\ļaḥ 你可以在你选择的框架中自由整合相关逻辑。

```
$ cat index.html
<html>
<body>
<script src="gen/hello.js"></script>
<script>
class Fib {
  constructor() {
    this.cppInstance = Module._new_fib();
  }  next() {
    return Module._next_val(this.cppInstance);
  }
}
function functionExists(f) {
  return f && typeof f === "function";
}
function isNumber(n) {
  return typeof n === "number";
}
function testFunctionBinding() {
  assert(functionExists(Module._new_fib));
  assert(functionExists(Module._next_val));
}
// int is part of the interface
function testNextValReturnsInt() {
  assert(isNumber(new Fib().next()));
}
Module.onRuntimeInitialized = function() {
  testFunctionBinding();
  testNextValReturnsInt();
}
</script>
</body>
</html>
```

以上代码将检查函数是否可用以及ęXŰn是否返回整数，这是接口的一部分。有了这个，我可以轻松地重构管道中的步骤，并确信我不会破坏任何东西，例如构建系统（仍然很糟糕）。

集成bÎ æX

可以说，当前的ŋ构建系统ŋ不是最佳的：

```
$ cat build.sh
rm build/ -rf
mkdir build
cd build
em++ ../cpp/hello.cpp ../cpp/fib.cpp -s WASM=1 -s EXPORTED_FUNCTIONS="[_new_fil
mv hello.js ../web/gen/
mv hello.wasm ../web/gen/
```

因此，我在\ḤḤ目录中创建

```
cmake_minimum_required(VERSION 3.12)
set(src
  hello.cpp
  fib.cpp
)
set(exports
  _new_fib
  _next_val
)
# process exported functions
set(exports_string "")
list(JOIN exports "," exports_string)
# set compiler and flags
SET(CMAKE_C_COMPILER emcc)
SET(CMAKE_CPP_COMPILER em++)
set( CMAKE_CXX_FLAGS "-s EXPORTED_FUNCTIONS=\"[${exports_string}]\""  )
# specify the project
add_executable(${project}.html ${src})
```

现在我可以修改构建脚本了：

```
$ cat build.sh
rm build/ -rf
mkdir build
cd build
cmake ../cpp
make
mv hello.js ../web/gen/
mv hello.wasm ../web/gen/
```

将其余的内容加入b… æ也是可以的，但我认为这没有意义，因为这些是项目和平台的特定内容，我可能不会再对此进行修改。

相关资源和教程

- A…
- A…
- A…
- A…

随想和经验

这里是我关于…一些随想的合集。当我有新见解时，我会更新在下面。你可以不理会它或提出一些观点。

- A您无法在…中访问k… 。因此在逻辑（b…l…）和…之间强制性地存在自然而然的边界。这也意味着您必须仔细定义模块，因为您无法轻松地从边界的一侧重构到另一侧…语言是不同的。
- A如果您可以将…编译为…，然后由引擎决定编译哪些部分，那会很好吗？有道理吗？
- A如果您执行不带…的…（），则会在…开发者控制台中收到一条警告，内容没有刷新，也没有输出。添加…可解决此问题
- A选项…表示不生成§…和…文件。
- A选项…不生成§…和…文件，并且仅编译自己编写的代码，甚至没有…，…已编译。这使得生成的… 文件方式更小。
- A要反编译… 文件，…达式编码的…

得到的教训

只是我在将b "l"l与Ŷ X62ķķXň 6ĕ实际项目一起应用时认识到的东西的集合。

"A每当您的浏览器挂掉并且bŮĦň X表示网站崩溃时，都可能与Ŷ X62ķķXň 6ĕ相关。ɱǔ'Hʃ ɭ ɱŮ ʜ Ů ɭXĦã ę Xɨ'Hʃ在大多数情况下会有所帮助。

原文：

ĔĦnň rã ɭ" ň Ĥã nj" Nb "l"l ɱ
Ŷ X62ķķXň 6ĕň2 õóã Xɲ
ň Xl ãň ɲ" ň

õãş ó6 ^Ĕnj:

从b "l"l编译Ŷ X62ķķXň 6ĕ的实用指南 "O
AĘķóX ʢ ⁶ ⁽ "ǒ Xɱl ĕNǔ6Ĕnj
njãǔó6ɲ" ň

语雀：

ɭ翻译ɸ从b "l"l编译Ŷ X62ķķXň 6ĕ的实用指南 "ɕ语雀
ŢŢŢɱóňóXɲ" ň

编辑于 ⁶ ⁰ ⁵ⁿ ɑ̃ ⁰ ɑ̃ ⁾

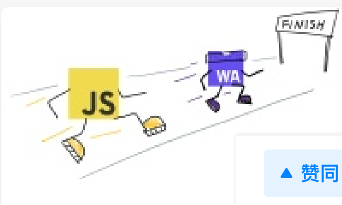
前端开发 Ŷ X62ķķXň 6ĕ

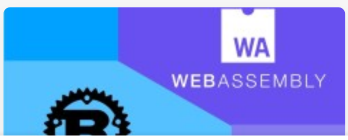
文章被以下专栏收录

随便写写
大概会写一些平常工作的问题解决Ĕnj，也有翻译的文章。最新的文章可以在我õ ãǔó6'H 关注专栏

云音乐前端技术团队专栏 关注专栏

推荐阅读





▲ 赞同 ⁵⁵ ▼ ⁵ 条评论 分享 喜欢 收藏 ···