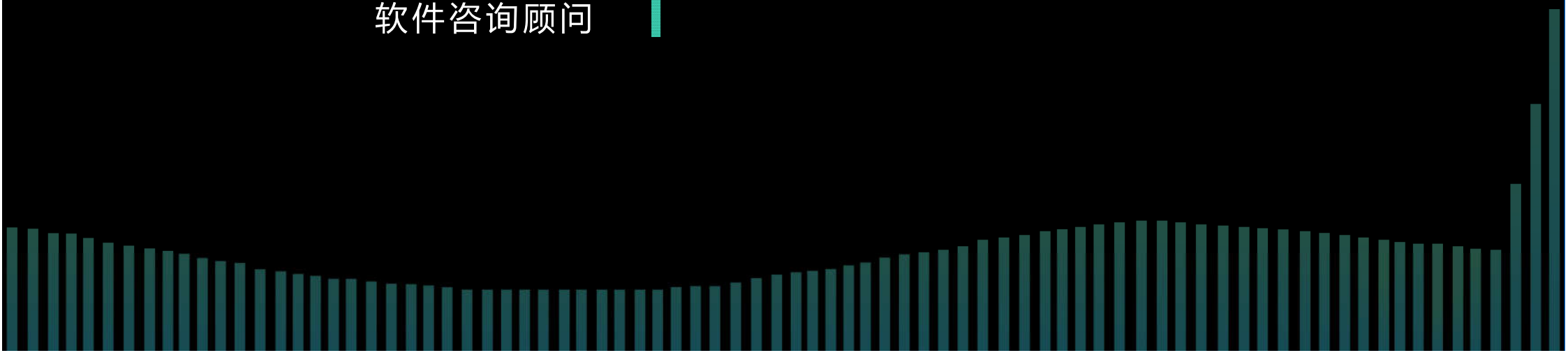
CPP-Summit 2020

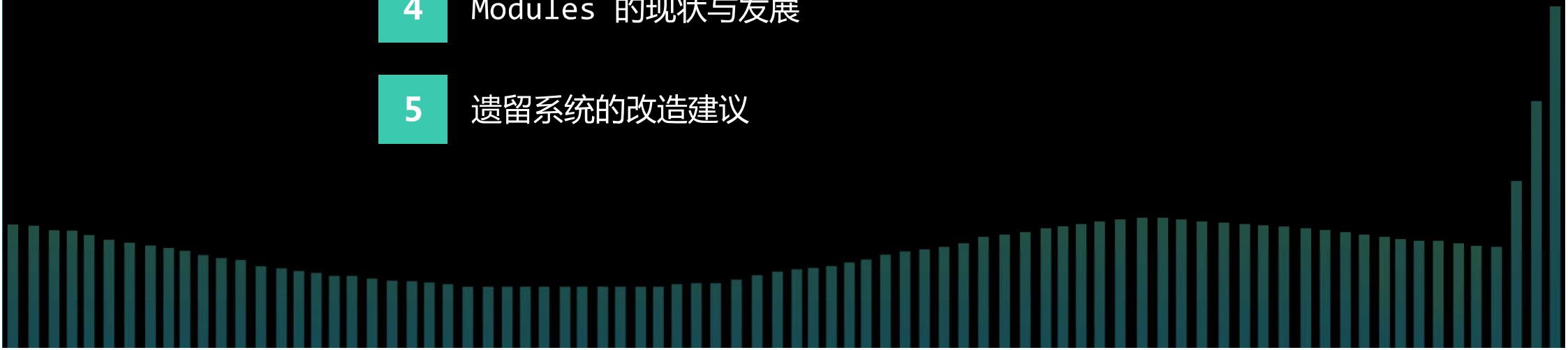
C++ Modules 与大规模物理设计

张超
软件咨询顾问



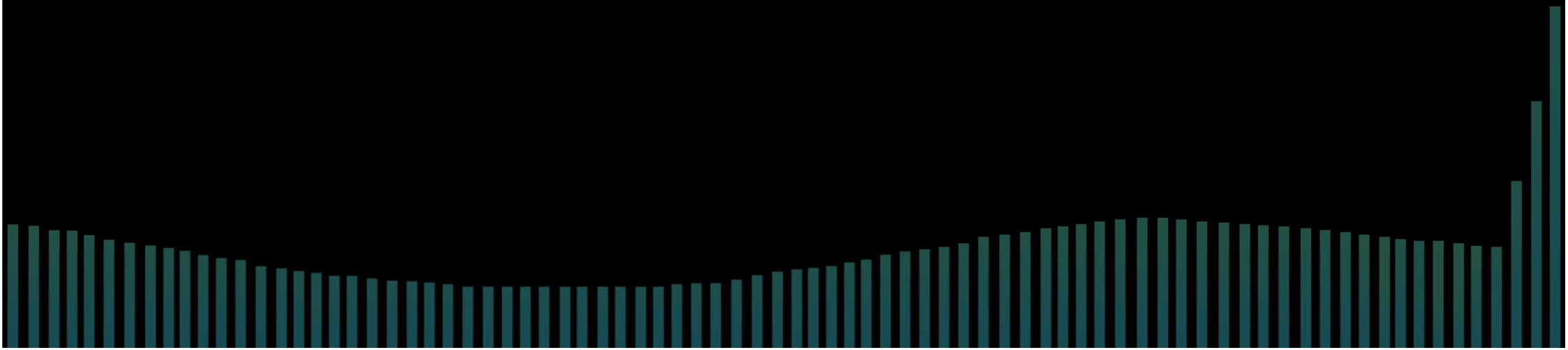
议程

1. C++ 物理设计的困境
2. C++ Modules介绍
3. Modules 改善的物理设计
4. Modules 的现状与发展
5. 遗留系统的改造建议

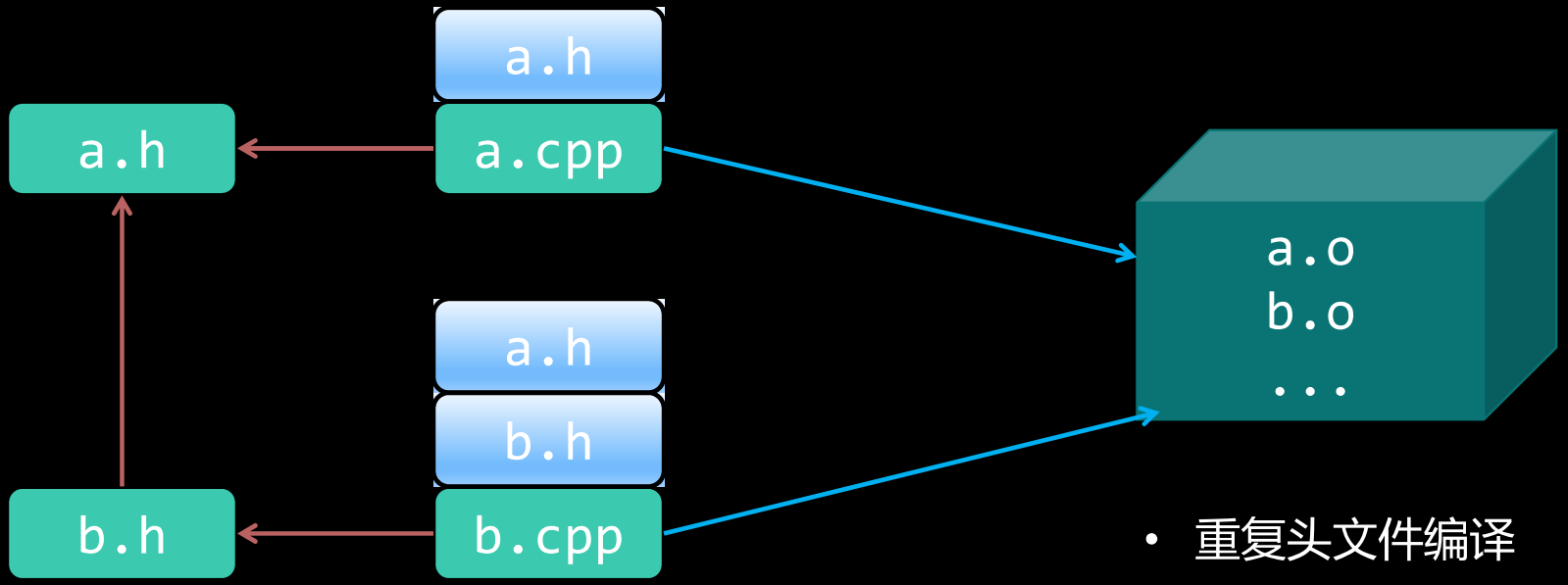


01

C++物理设计的困境

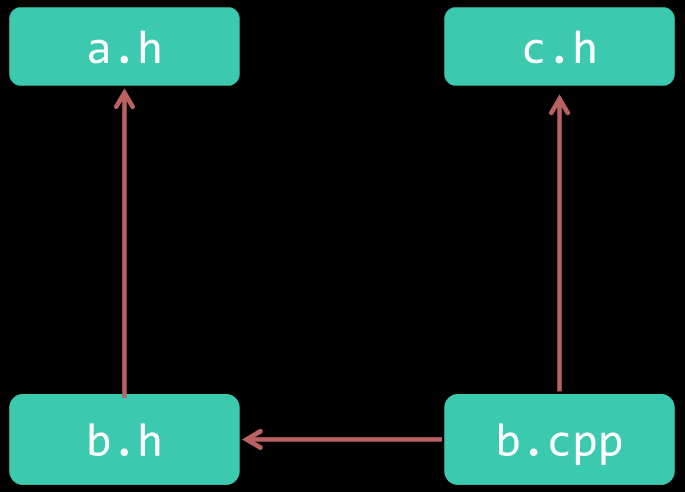


编译时间长



- 重复头文件编译
- 链接无序查找

可能对程序造成的错误和干扰

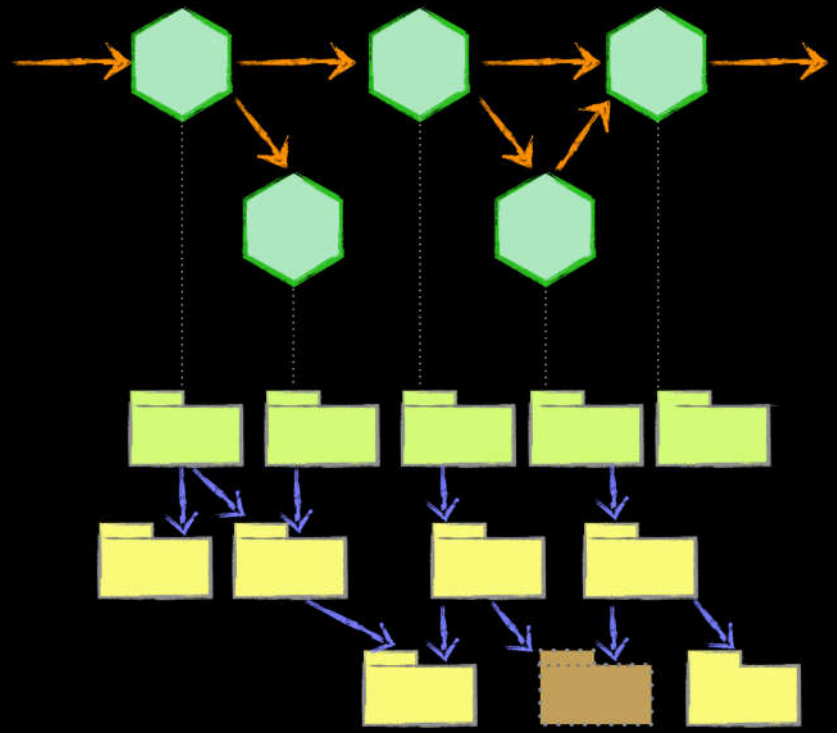


- 头文件不自满足
- 宏定义经由头文件传播导致的错误替换
- #include 的顺序导致解释差异
- 符号的重名

难以理清的依赖关系

- A include B 不知道具体用了哪些东西，
- A include B 不知道间接依赖其他的头文件
- A link B 只在构建脚本中体现
- Extern 的使用让依赖关系更无处可寻

物理设计的目标

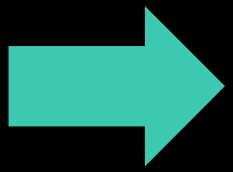


- 编译：可以高效正确的执行
- 开发：更好的服务于开发效率，不容易出错
- 架构设计：直观的反映逻辑设计和依赖关系

物理设计困难根因分析

现有手段 ：

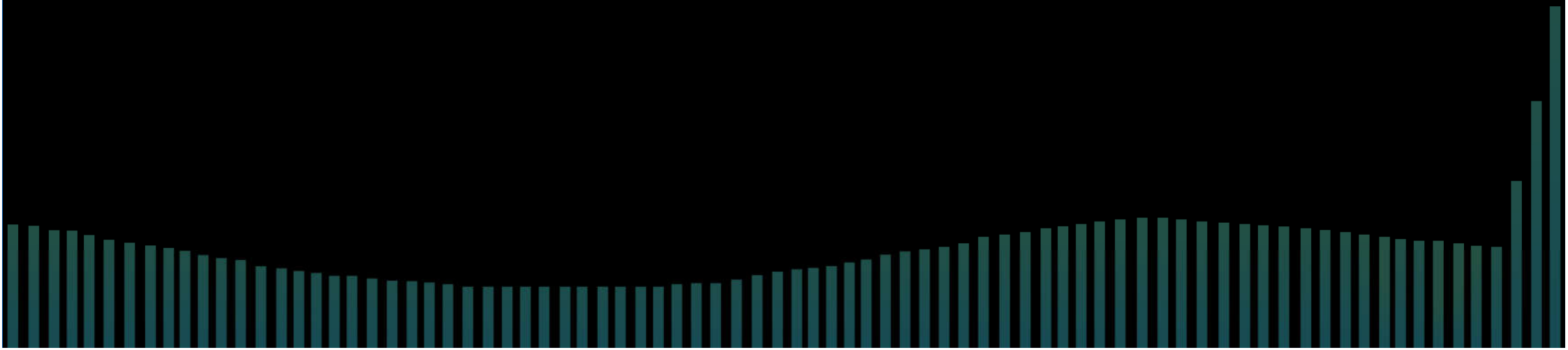
- 编译时间长：物理设计原则，PCH
- 宏定义破坏：无能为力
- 符号冲突：static/namespace
- 依赖管理：物理设计原则



- 预处理机制存在缺陷
- 缺乏可见性控制手段

02

C++ Modules介绍



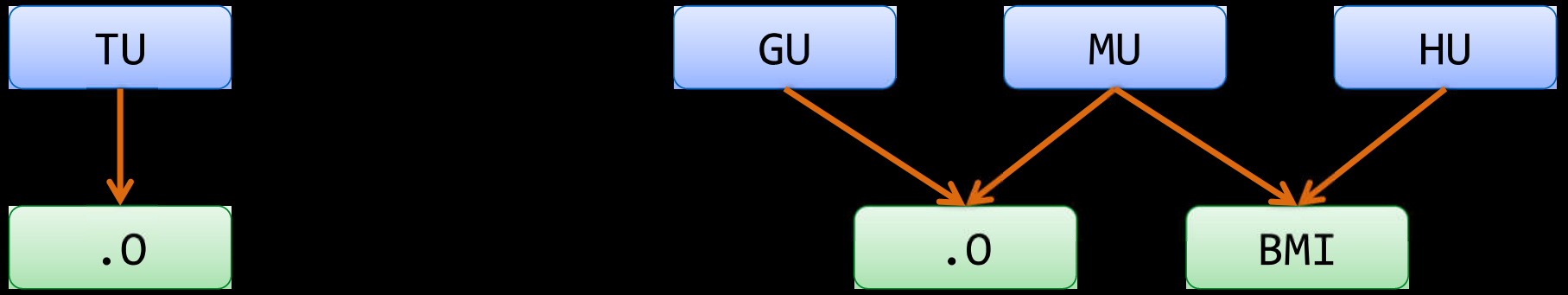
MODULES 的解题思路

- 新的引用方式import与新的复用单元module代替#include与头文件
- 提供显示的控制符号的能力export

MODULES 新增概念

- Translation Unit

- Module Unit: cppm/ixx/mxx/
- Global Module : cpp/cxx
- Header Unit: h/hpp (importable)
- BMI (Binary Module Interface) /CMI



关键字-MODULE

```
// TU 1
export module A;
export int baz();
// TU 2
export module Foo;
export int foo() {
    return 0;
}

// TU 3
module;
#include <math.h>

module A;

int bar();
int baz() {
    return abs(bar()) + 1;
}

module :private
int bar() {
    return -1;
}
```

- 声明Module
- 全局片段
- 实现Module（可选）
- 私有片段

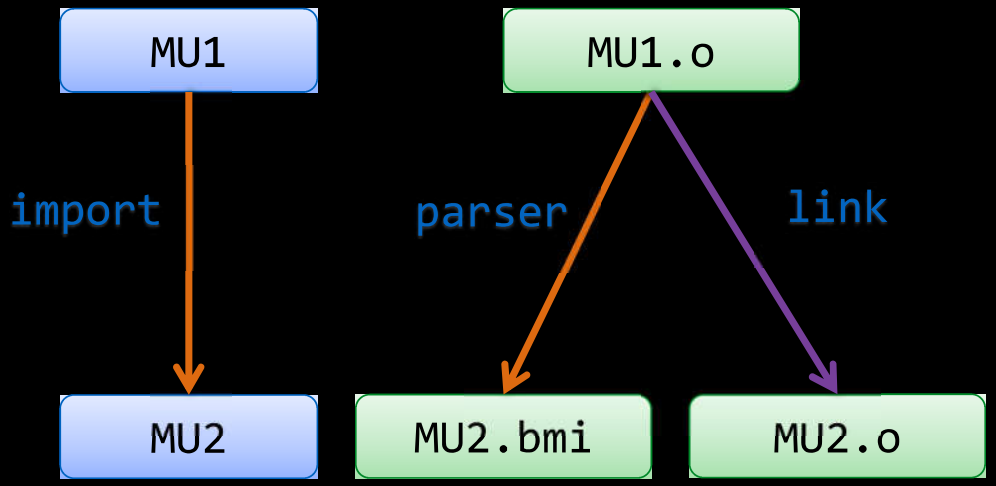
- Module的范围从声明（定义）开始
- Module的合法名字使用“.”连接
- 全局片段中的内容，全局链接
- 私有片段里的内容，不跨MU链接

关键字-EXPORT

```
module A;
export int bar() {
    return -1;
}

export {
    class e1 {
        int x;
        int y;
    };
    int e1;
}

namespace baz {
    export void quux();
}

export namespace bar{
    void f1();
    int v1 = 0;
}
```

- 必须有名字的才export（编译过后有名字的）；static函数，匿名namespace，不能导出
- export namespace 导出 本段 export namespace 包含的内容
- export namespace 中的部分，间接导出命名空间名字
- 支持 export block
- Module Implementation 不能包含export

关键字-IMPORT



import A::B 不是A中的B内容，A::B 一定也是个MU

import 也有循环依赖的风险。

import 不等于#include

import 2个不同module中相同的符号，仍然会冲突

import 只能在全局范围，不能在namespace里

一个MODULE UNIT的完整例子

```
//Global Fragment
modules;
#include "some.h"

//ModuleName
export module M;
export module :Internal

//Imports
import X;
import Y;

//Exports Internal Depends
struct FooBase {
};

//Exports
export struct Foo : FooBase{
};

// Private Fragment
modules:private;

//Internal Implementation
int xx(){
}
```

- 建议只放include
- 声明Module
- Import
- Export内容的依赖声明
- Export
- 内部实现

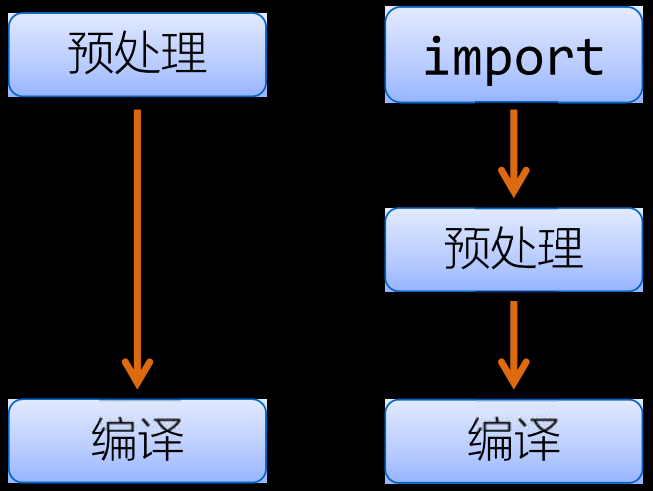
MODULE UNITS 分类

```
// TU 1
export module A;
export import :Foo;
export int baz();
// TU 2
export module A:Foo;
import :Internals;
export int foo() {return 0;}
// TU 3
module A:Internals;
int bar(){ return 1;}
// TU 4
module A;
import :Internals;
int baz() { return bar() + 1;}
```

- Module Interface
- Module Implementation
- Module Partition

- 方便演进
- 进一步降低依赖
- 并行开发

MODULE 与 MACRO



- 宏的作用范围不会超过module。
- 需要共享的宏定义只能在Header Unit中，不可传递
- 不要用宏定义modules的关键字

MODULE 重定向

```
// TU 1
export module FooWrapper;
export import Foo;
export import Foo1;
export import Foo2;
export import Foo3;

// TU2
export module Foo;
export import Foo:doo;
```

聚合

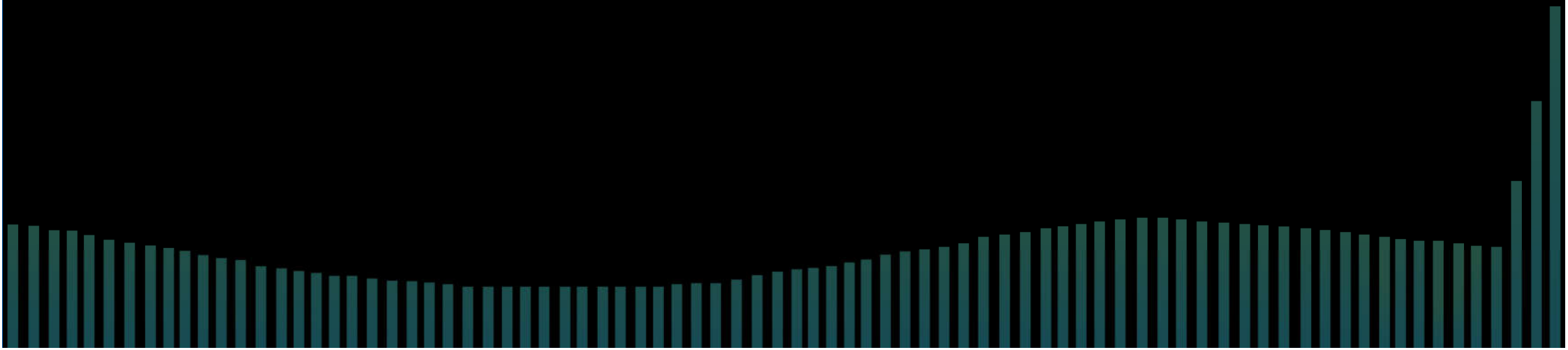
常用作库的接口，方便并行开发后整合。

封装

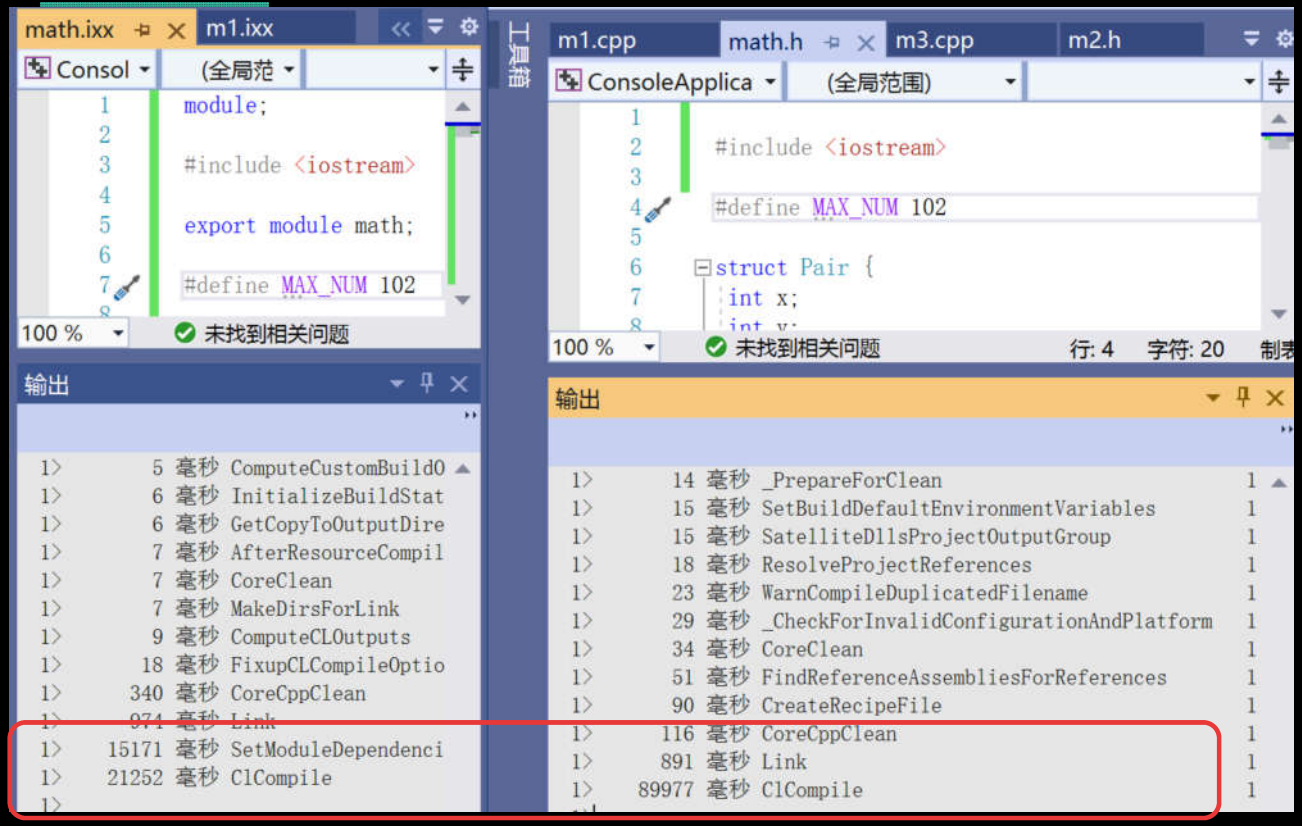
为了特定用户封装部分接口；
非接口module最好不要
export import

03

Modules改善的物理设计



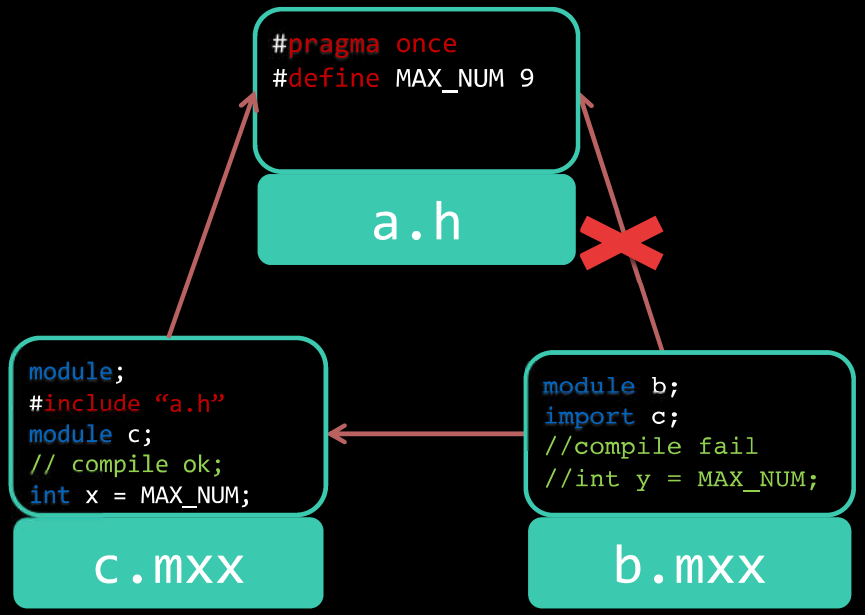
MODULES 带来编译速度提升



- 减少了头文件的重复编译；
- 确定方向的链接；
- 对于头文件很小，不包含标准头文件的系统，或已经PCH/UNIT SOURCE优化的，提升有限。

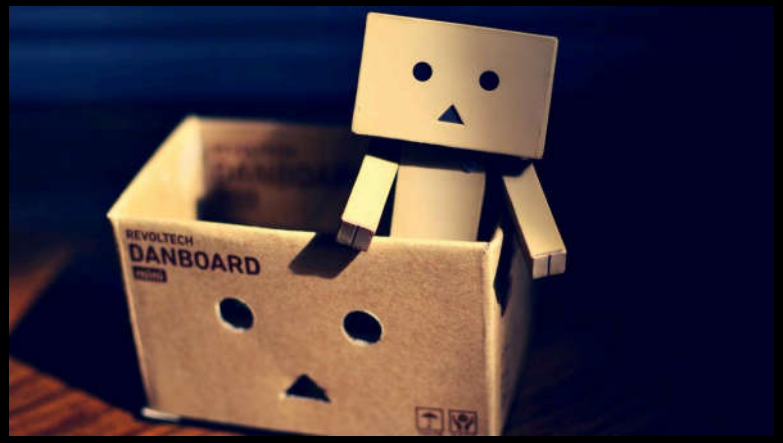
“100个cpp 包含 （include）1个头文件 ” vs “100个ixx 引入（import） 1个ixx”

MODULES 为依赖管理带来的改善



- 宏定义不能跨Module传递
- 可以轻松从源码获取的准确的依赖关系
- 选择性可见，选择性可达
- 符号链接有了确定的指向

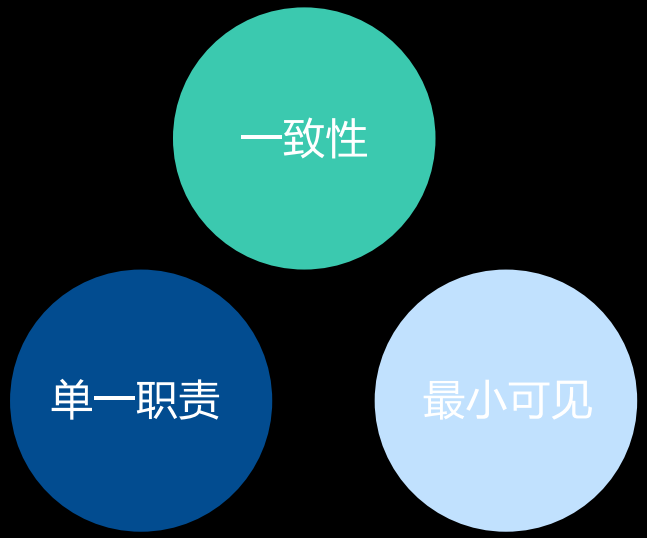
MODULES 带给包管理的机遇和挑战



- 大大提高了接口包的稳定性
- 源码与二进制之外，新的复用单元（BMI）
- 代码内置的依赖关系，需要与描述文件中的内容一致，最好避免重复声明
- Modules的命名，目录应该与包管理路径的结合。

MODULES没法代替你做出良好的物理设计

物理依赖与逻辑依赖保持一致

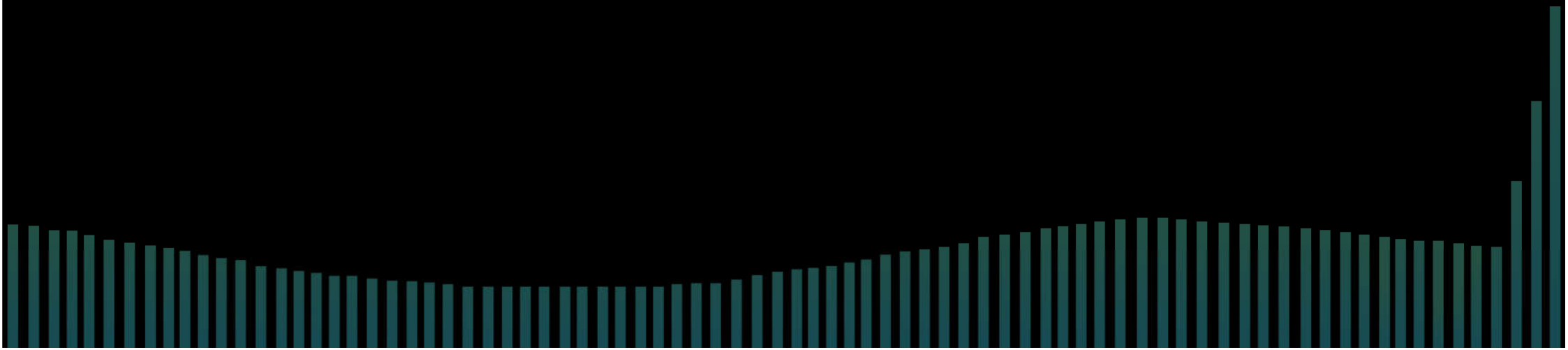


只有一类导致模块变化的原因

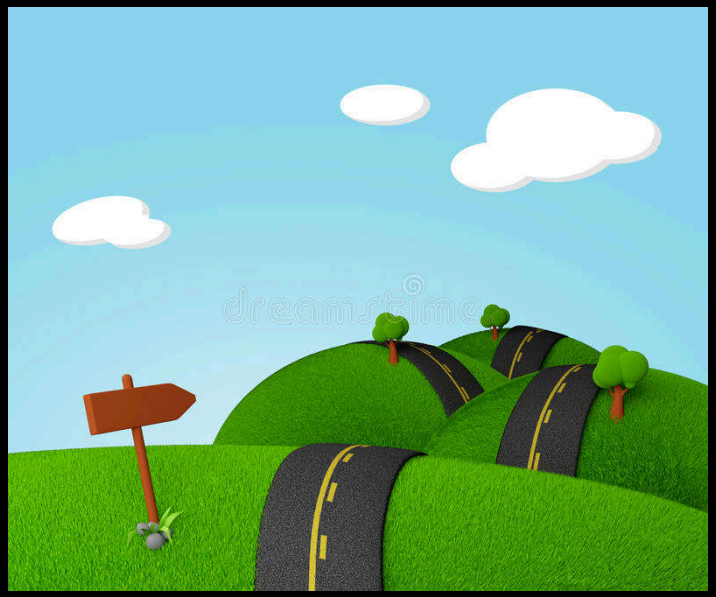
尽量少的暴露接口和结构

04

Modules的现状与发展



MODULES 编译器标准现状



- 编译器：clang，gcc(branch)，msvc
- 构建工具：cmake，ninja，build2…
- IDE：eclipse，clion，vs2019…

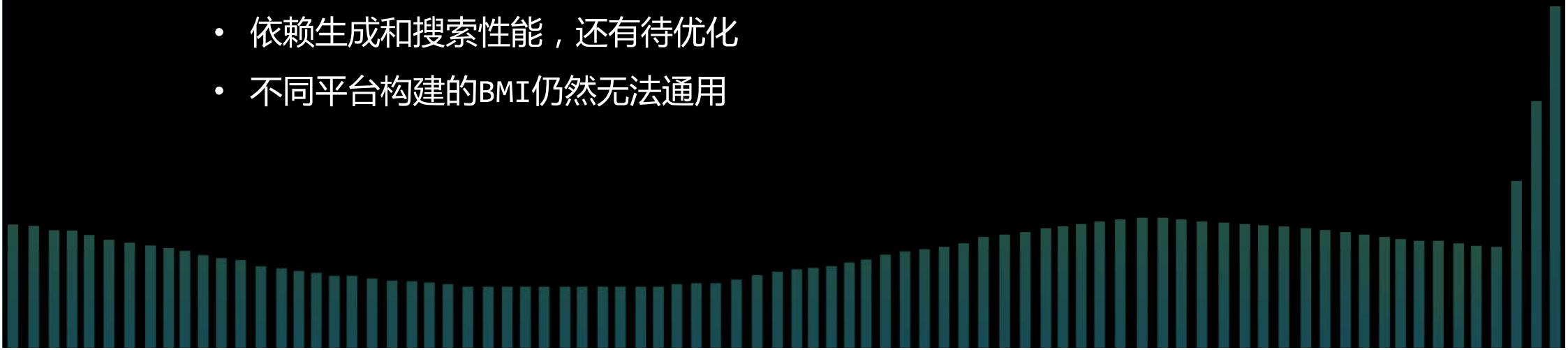
Modules对整个C++的生态链都是颠覆式的改变

MODULES 对比其他语言

语言	模块定义	导出	导入	模块描述文件
C++20	module	export	import	内置
python	目录/文件	all	import	外置.__init__py
rust	mod	pub	use	内置
golong	package	all	import	外置.mod
java	module/package	exports/all	requires/import	module-info.class

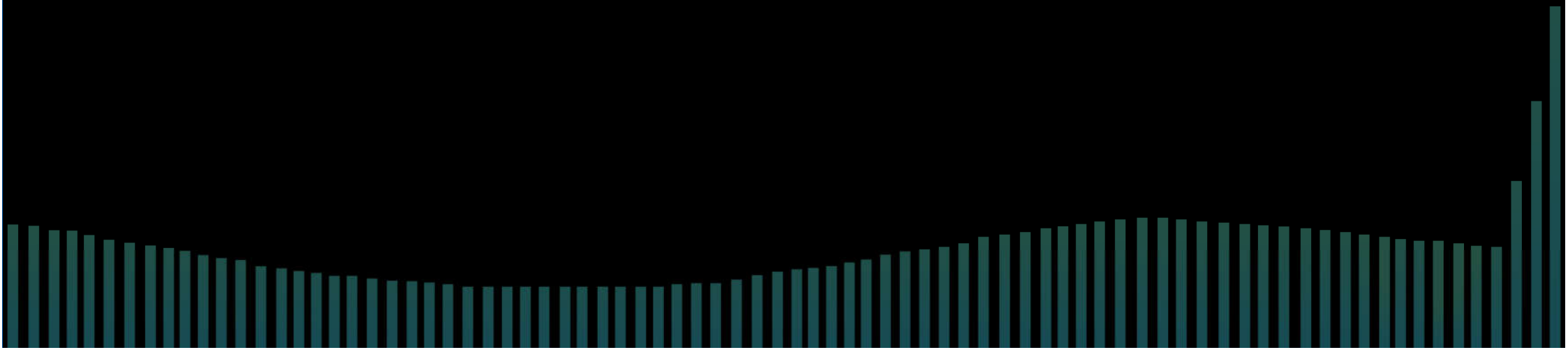
MODULES 遗留问题

- 灵活的命名带来的困扰：
  - 可以多个“.”连接，但没有层次的语义，
  - module的名字与文件名没有任何关系。
- 不支持单个符号的import，
- 无用的import的可以优化处理
- 依赖生成和搜索性能，还有待优化
- 不同平台构建的BMI仍然无法通用

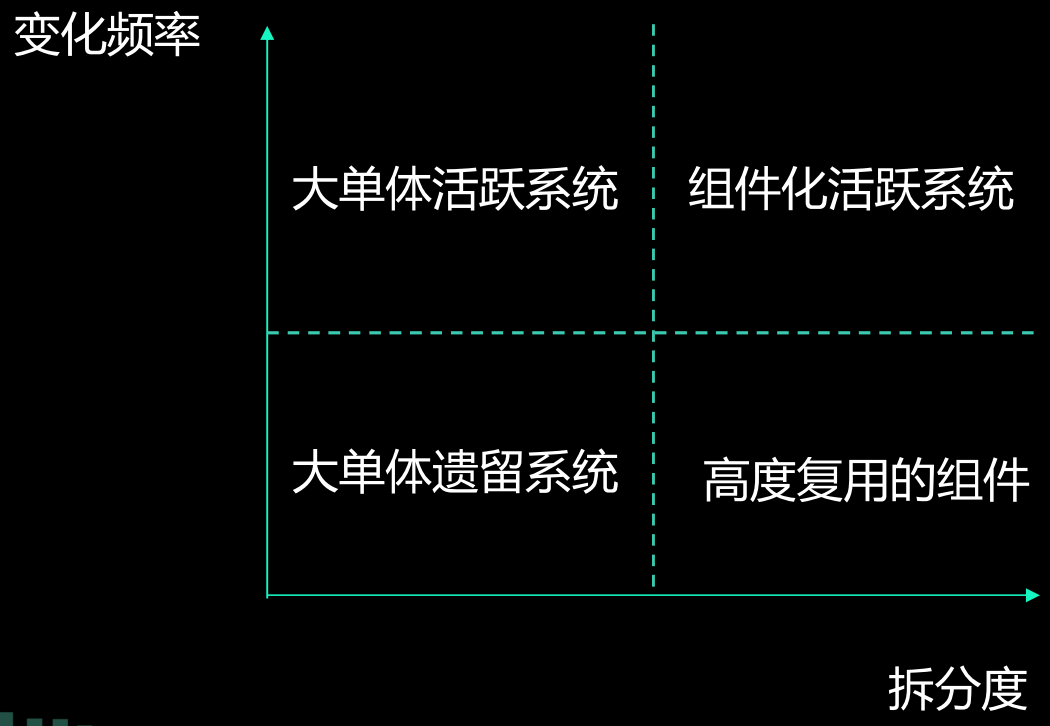


05

遗留系统的改造建议



如何演进到MODULES 的一些原则



原则：

- 适可而止
- 小步安全的重构

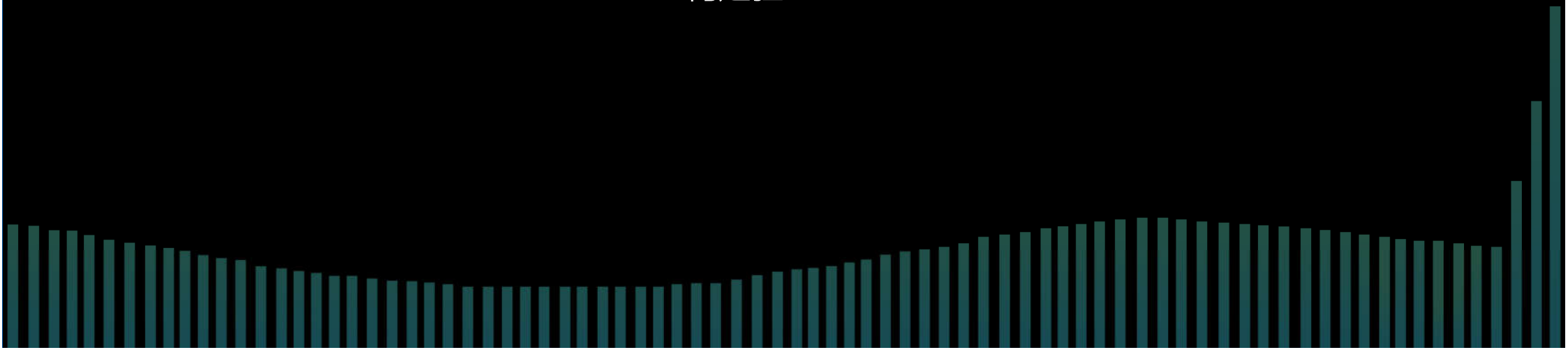
场景1:大型单体遗留系统-建议编译提升

分析：

- 规模大重构风险高
- 需求少验证成本高

重构步骤：

1. 替换或拆分头文件为Header Unit(importable)
2. 替换依赖它的原有inlcude为import，或增加新的import
3. 构建验证



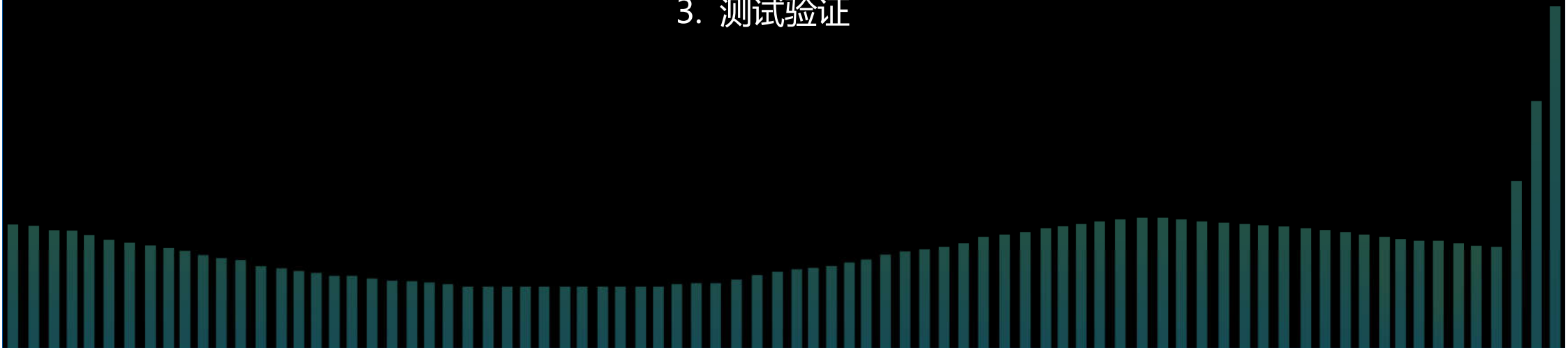
场景2:大单体活跃系统-建议MODULE拆分

分析：

- 需求活跃，可随需求验证
- 生命周期长，对响应变化有诉求

重构步骤：

1. 找到可复用的单元，逐步提取出到新的Module文件中，export相应的接口。
2. 原有的代码需要用到Module的，import 新的Module。
3. 测试验证



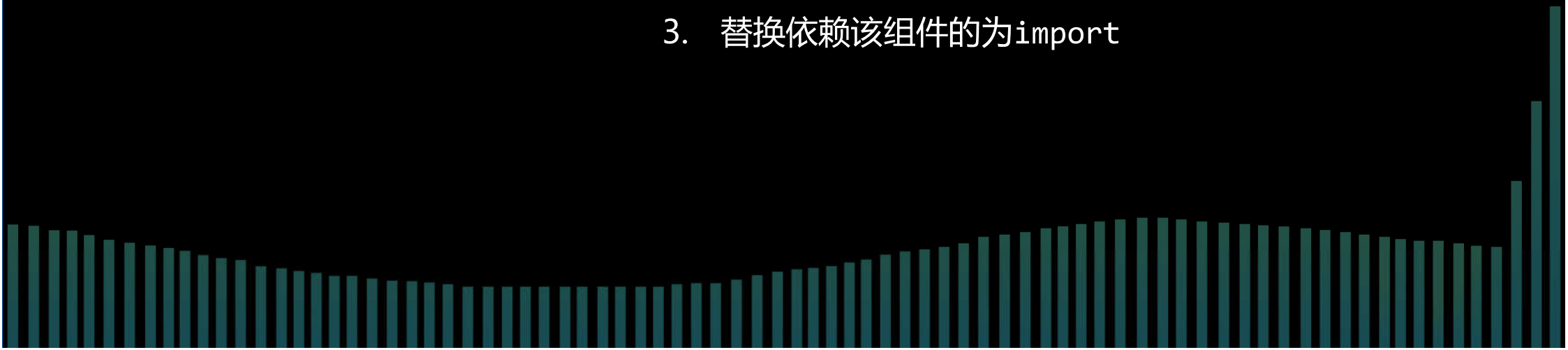
场景3:组件化活跃系统-建议合理化依赖

分析：

- 组件间依赖可控，依赖双方可同时修改
- 有相对清晰的边界

重构步骤：

1. 修改include文件为Module Interface，export相应的接口，cpp改为Module Implement。
2. 调整组件对外部库的inlcude为聚合Module Interface。
3. 替换依赖该组件的为import



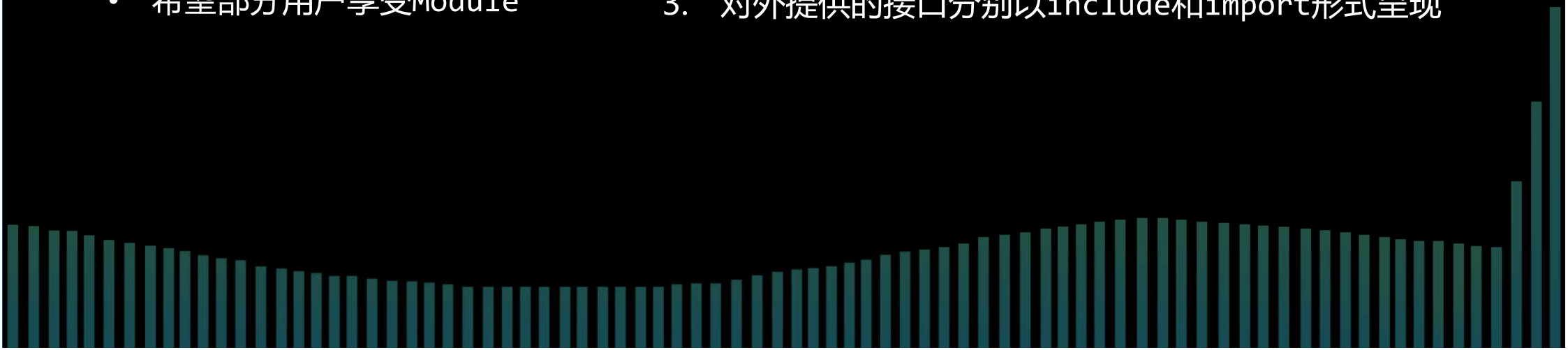
场景4：高度复用的组件-并存过渡接口

分析：

- 对库的使用者没法控制同步修改
- 希望部分用户享受Module

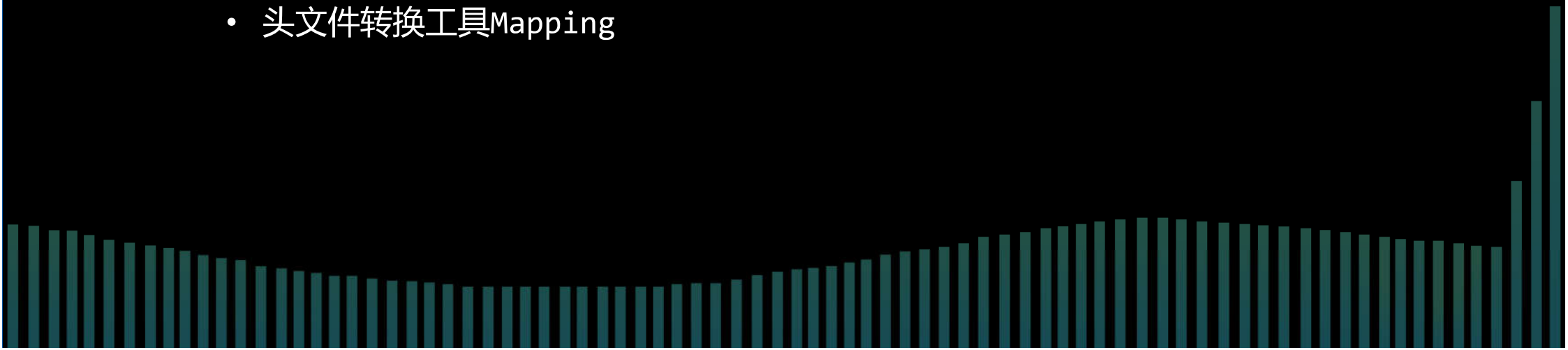
重构步骤：

1. 替换或拆分头文件为Header Unit(importable)
2. 实现文件全部为Global Unit。
3. 对外提供的接口分别以include和import形式呈现



MODULES大势所趋，理当积极准备

- 好的物理设计距离Modules之差一步之遥
- 准备好宏定义，按照Modules风格设计文件，切换编译器时搜索替换
- 保持Module文件命名和路径与文件系统保持一致
- 借助包管理工具熟悉并设计Module的层次以及梳理依赖关系
- 头文件转换工具Mapping



谢谢