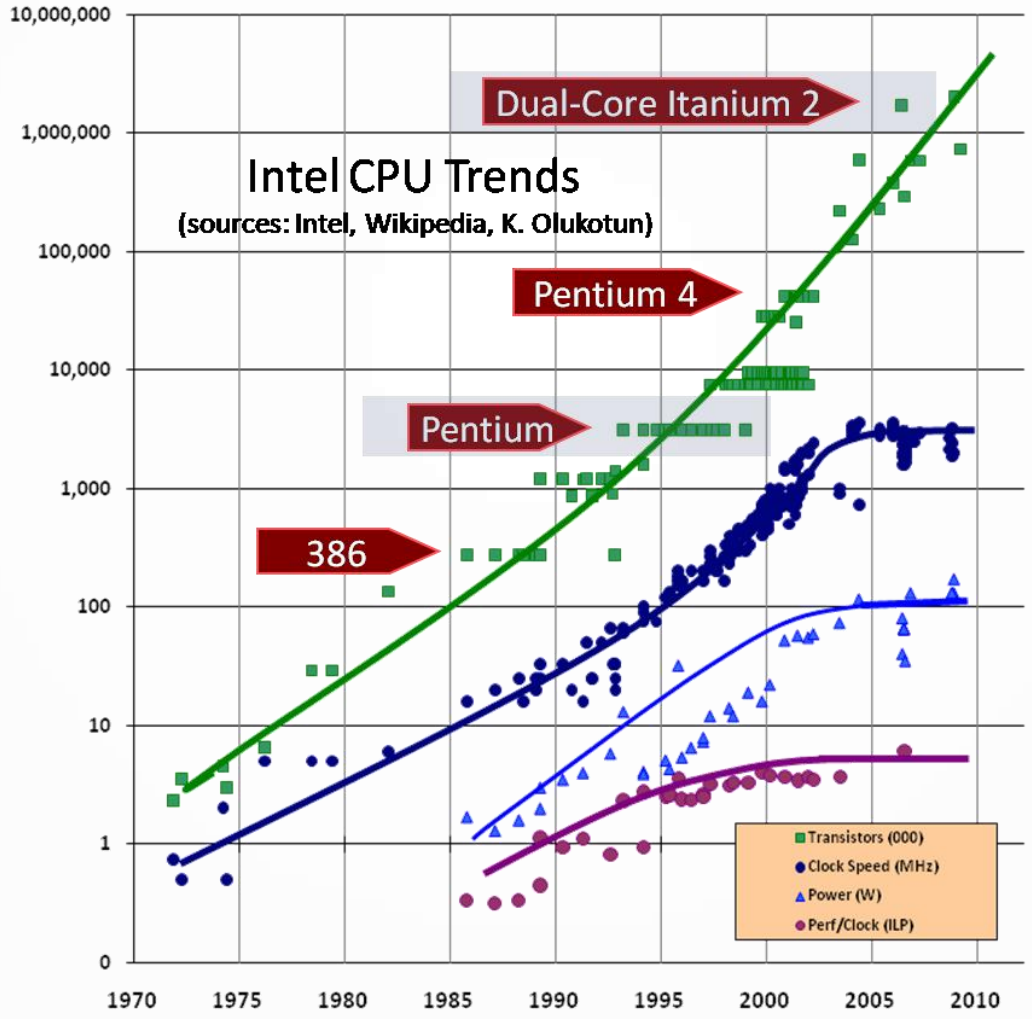
「基础设施与软件架构」专场

从 C++ 新特性的实现与实践谈起：C++ 生态的发展趋势及其影响

许传奇 2023年05月28日

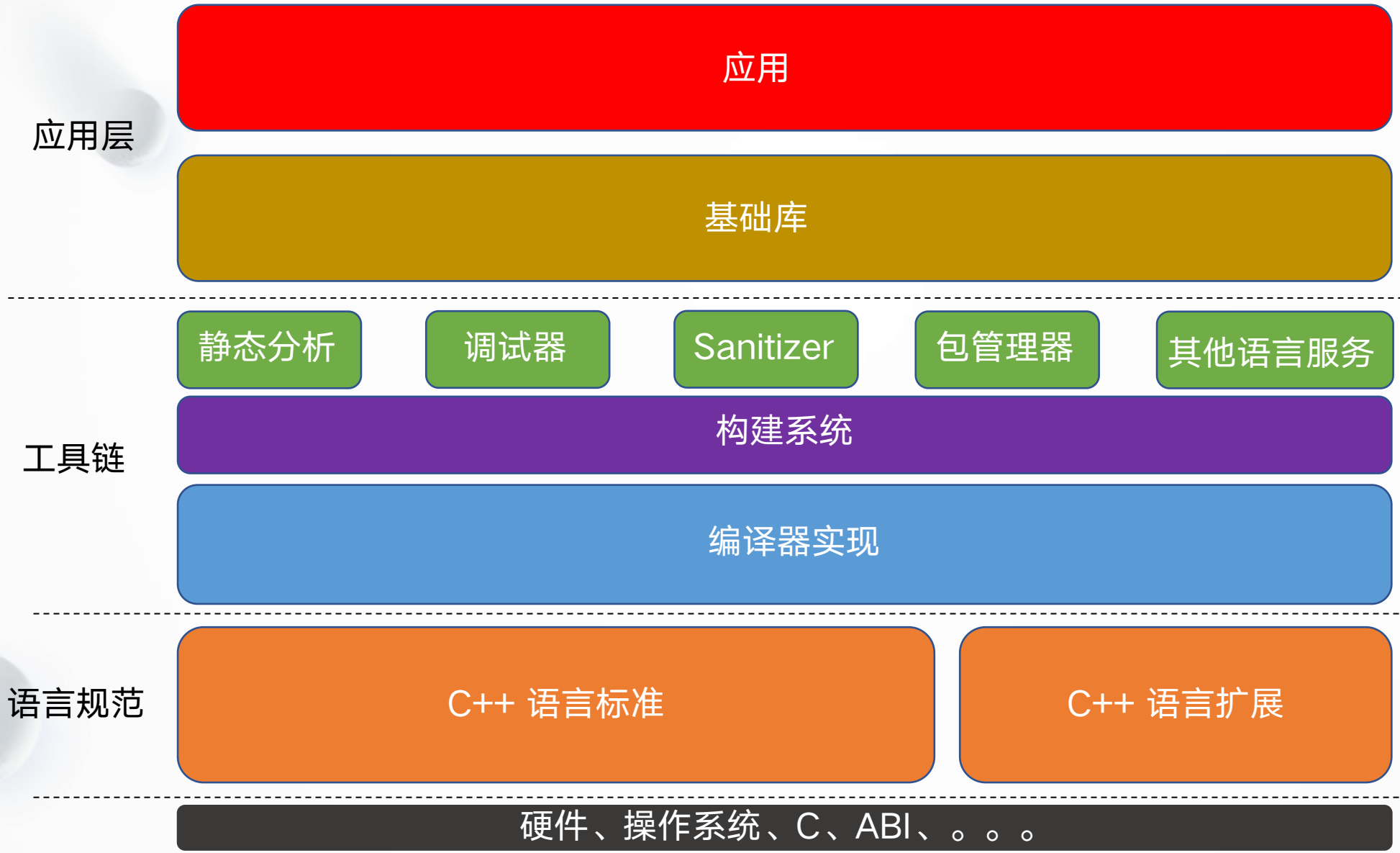
C++ 成为 TIOBE 2022 年度语言

Year	Winner
2022	C++
2021	Python
2020	Python
2019	C
2018	Python
2017	C
2016	Go
2015	Java
2014	JavaScript
2013	Transact-SQL
2012	Objective-C
2011	Objective-C
2010	Python
2009	Go
2008	C
2007	Python
2006	Ruby
2005	Java
2004	PHP
2003	C++



The free lunch is over

Herb Sutter


应用层
应用
基础库
工具链
静态分析
调试器
Sanitizer
包管理器
其他语言服务
构建系统
编译器实现
语言规范
C++ 语言标准
C++ 语言扩展
硬件、操作系统、C、ABI、。。。

C++ 生态发展趋势

- 应用层：异步化与并行化
- 语言标准、底层库：编译期计算
- 工具链：围绕 Modules 建立 C++ 新生态
- 语言标准、工具链、应用：安全

异步化

异步开发一直是提高并发度的不二法门，但由于其复杂度以及 C++ 本身的复杂度。长久以来，在 C++ 中异步编程都不是一件容易的事。

异步化

但随着时间的发展，C++ 在库方面也有了例如 asio、seastar、execution 这样的成熟、易用的异步库。在语言方面更是引入了 C++20 协程这样的杀器。使得异步编程既简单也高效。

异步化

随着 asio 库、协程库的逐渐成熟，越来越多的 C++ 项目选择进行异步化改造。已经使用线程池、有栈协程等异步方案的 C++ 项目也有许多选择使用 C++ 协程以简化代码。

异步化

C++20 协程简介

- 协程是一个可挂起的函数。
  - 支持以同步方式写异步代码。
- C++20 协程是无栈协程。在语义层面不保存调用上下文信息。
  - 对比有栈协程
    - 更好的执行 & 切换效率。
  - 对比 Callback
    - 更简洁的编程模式，避免 Callback hell。

异步化

C++20 协程例子 – 同步例子

```
uint64_t ReadSync(std::vector<File> Inputs) {
    uint64_t read_size = 0;
    for (auto &&Input : Inputs)
        read_size += ReadImplSync(Input);
    return read_size;
}
```

异步化

C++20 协程例子 – 异步例子

```
template <RangeT Range, Callable Lambda>
future<void> do_for_each(Range, Lambda);                    // We need introduce another API.
future<uint64_t> ReadAsync(vector<File> Inputs) {
    auto read_size = std::make_shared<uint64_t>(0);         // We need introduce shared_ptr.
    return do_for_each(Inputs,                                                  // Otherwise read_size would be
                       [read_size] (auto &&Input){          // released after ReadAsync ends.
                           return ReadImplAsync(Input).then([read_size](auto &&size){
                               *read_size += size;
                               return make_ready_future();
                           });
                       })
        .then([read_size] { return make_ready_future<uint64_t>(*read_size); });
}
```

C++20 协程例子 – 协程例子

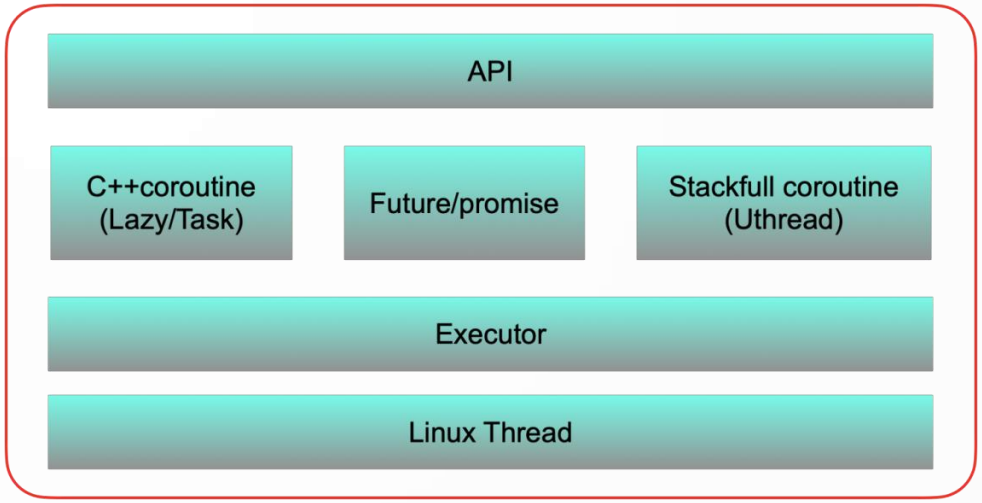
```
Lazy<uint64_t> ReadCoro(std::vector<File> Inputs) {
    uint64_t read_size = 0;
    for (auto &&Input : Inputs)
        read_size += co_await ReadImplCoro(Input);
    co_return read_size;
}
```

- 使用同步方式写异步代码。
- 兼具开发效率与运行效率。

异步化

C++20 协程库

- async_simple
  - 设计借鉴了 folly 库协程模块。
  - 轻量级。
  - 包含有栈协程、无栈协程以及 Future/Promise 等异步组件。
  - 从真实需求出发。
  - 与调度器解藕，用户可以选择合适自己的调度器。
  - 经受了工业级 Workload 的考验。
  - 开源于：https://github.com/alibaba/async_simple
- coro_rpc
  - 基于 async_simple 与 struct_pack 的高性能易用 RPC 库
  - 开源于：https://alibaba.github.io/yalantinglibs/zh/coro_rpc/coro_rpc_introduction.html



异步化

C++20 协程应用

- 业务1（1M Loc、35w core）
  - 原先为同步逻辑
  - 协程化后 Latency 下降 30%
  - 超时查询数量大幅下降甚至清零
- 业务2（7M Loc）
  - 原先为异步逻辑
  - 协程化后 Latency 下降 8%
- 业务3（100K Loc、2.7w core）
  - 原先为同步逻辑
  - 协程化后 qps 提升 10 倍以上性能

并行化

并行化是今日 C++ 应用提升性能的另一股潮流
例如 CUDA、OneAPI、OpenCL 和 OpenMP 等都是很火热的 C++ 方向

并行化

在 C++ 中相关的内容位于 Parallelism TS 中, 其中距离标准化最近的则是 SIMD 库:

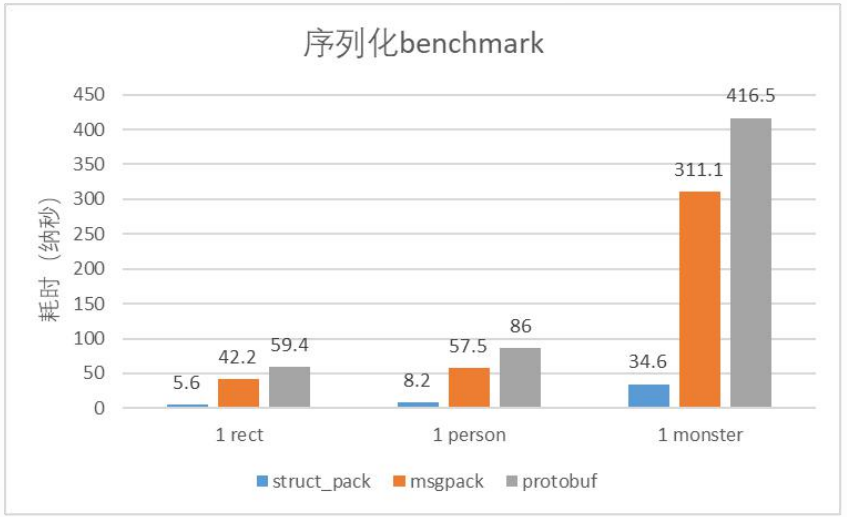
- SIMD 库已脱离 Parallelism TS，后续讨论将在 LEWG 中展开。
- SIMD 库会将各个平台的 SIMD Intrinsics 封装为标准接口。
- 使用标准库接口更友好。
- 标准库实现会比手写汇编效率更高。
- 标准库封装对编译器实现向量化有帮助。
- 对于有体系结构迁移的项目来说，SIMD 库会很有帮助。

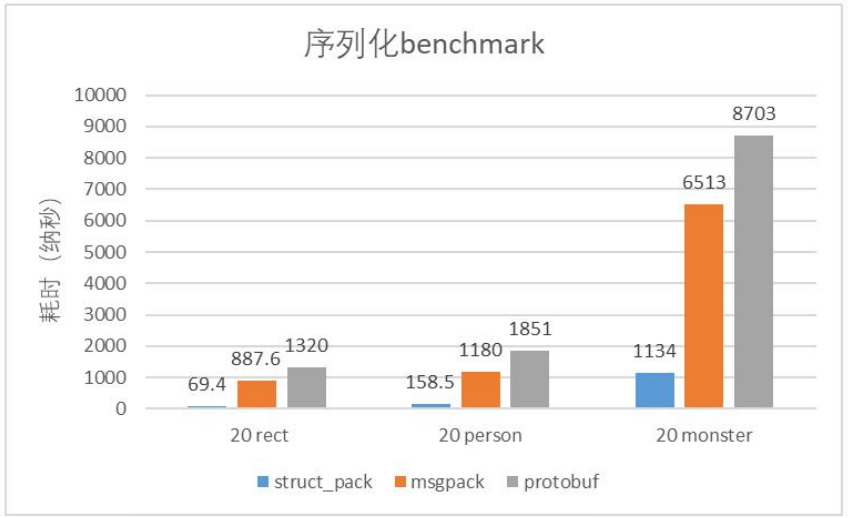
编译期计算

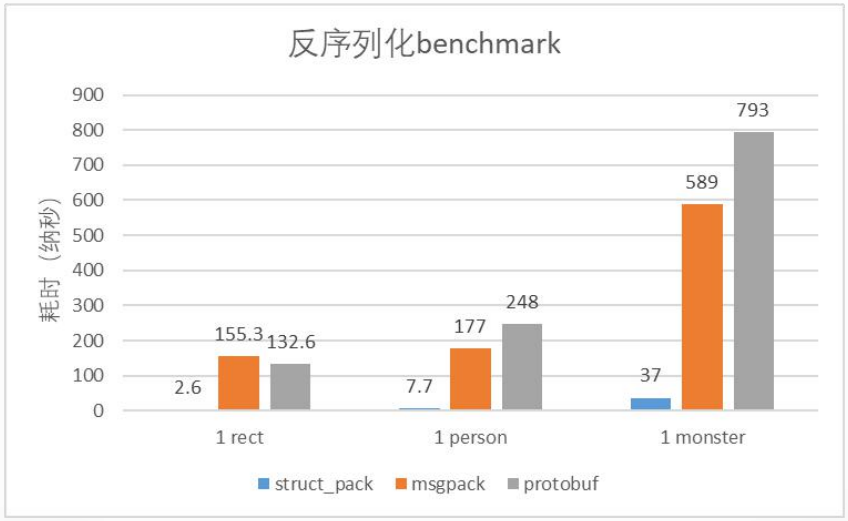
- 为尽可能提高效率将尽可能多的计算提前到编译期
- 近年来标准库演化的一大潮流便是为许多组件加上 constexpr
- 核心语言演化的一个主要潮流也是降低编译期计算、模版编程的复杂度
- 截止 C++20，我们已经可以通过现有语言能力，在库级别模拟反射的能力

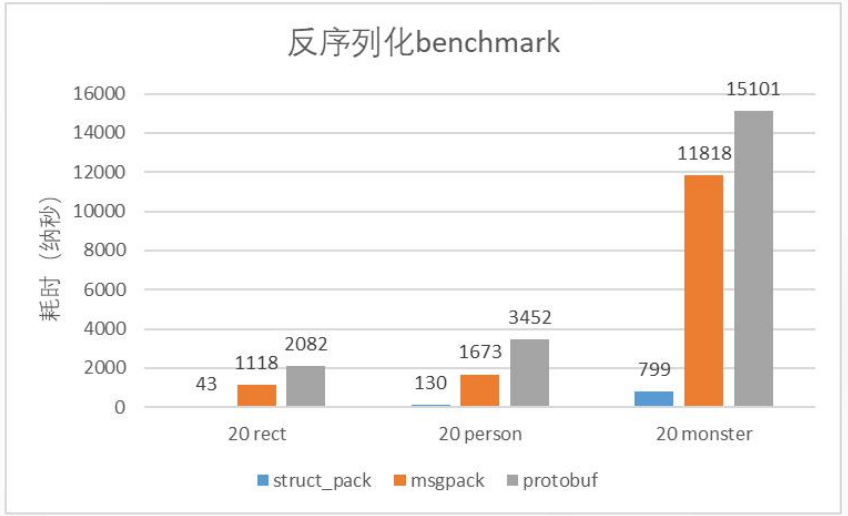
编译期计算

- struct_pack
  - 基于编译期计算的高性能序列化库
  - 在编译期完成 MD5 检验
  - 0 成本抽象
  - 针对不同类型做特定优化









Modules

Header files are a major source of complexity, errors caused by dependencies, and slow compilation. Modules address all three problems.

Bjarne Stroustrup

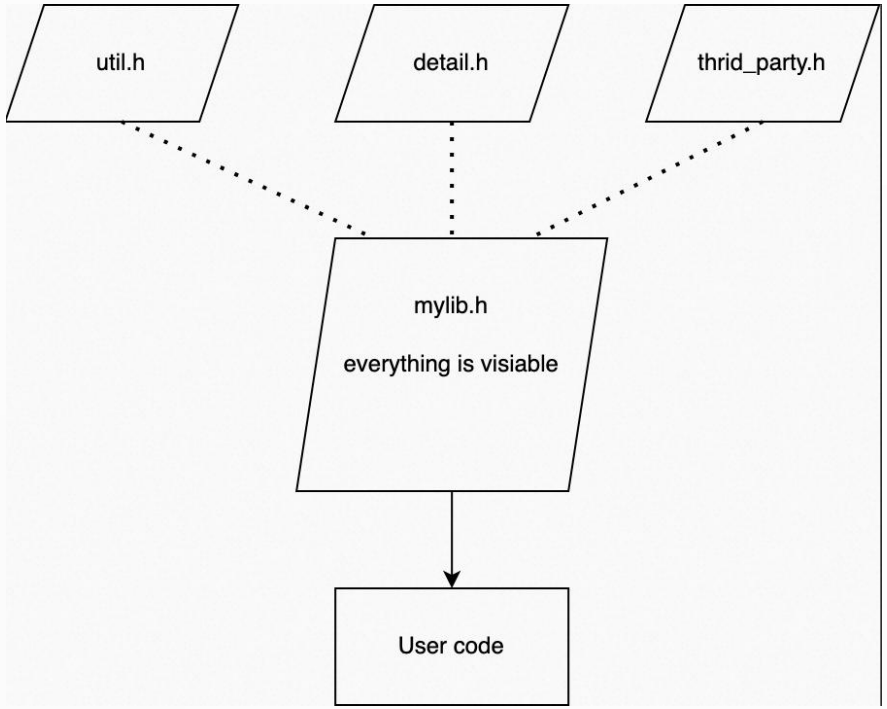
Modules

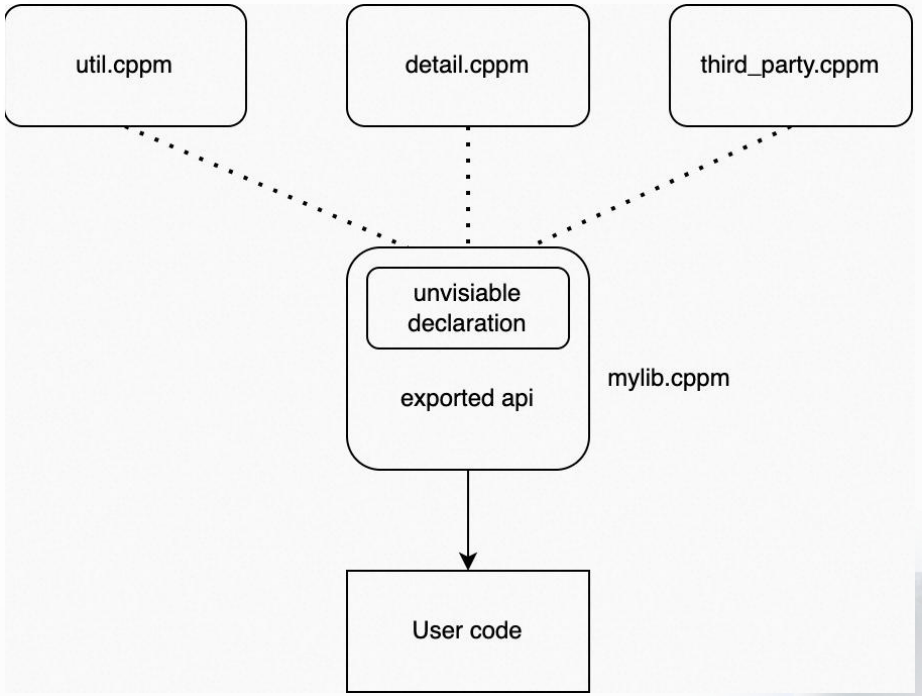
```
// Hello.cppm
module;
#include <iostream>
export module Hello;
export void hello() {
        std::cout << "Hello
World!\n";
}
```

```
// use.cpp
import Hello;
int main() {
    hello();
    return 0;
}
```

Modules

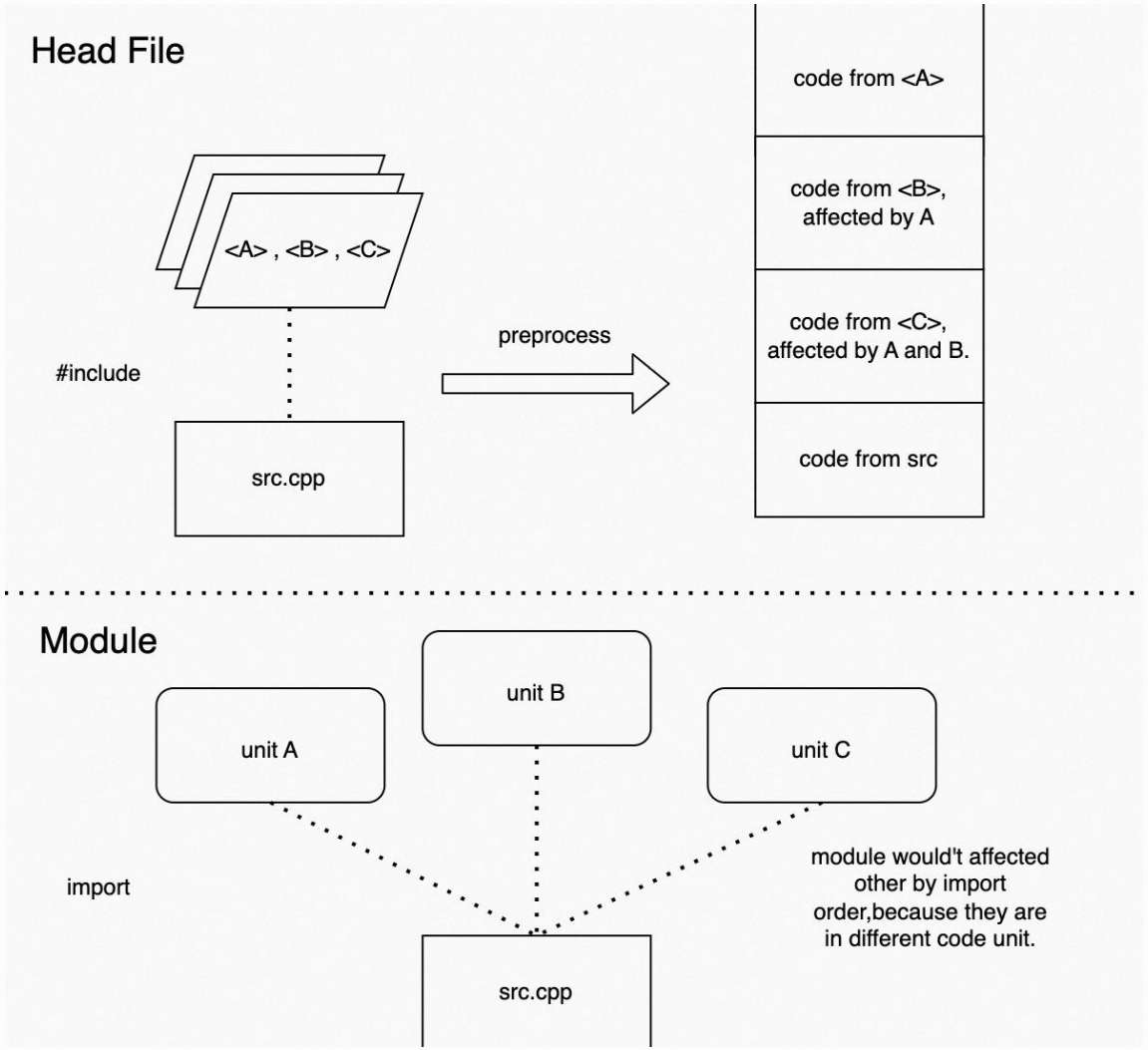
封装性


util.h
detail.h
thrid_party.h
mylib.h
everything is visiable
User code


util.cppm
detail.cppm
third_party.cppm
unvisiable declaration
exported api
mylib.cppm
User code

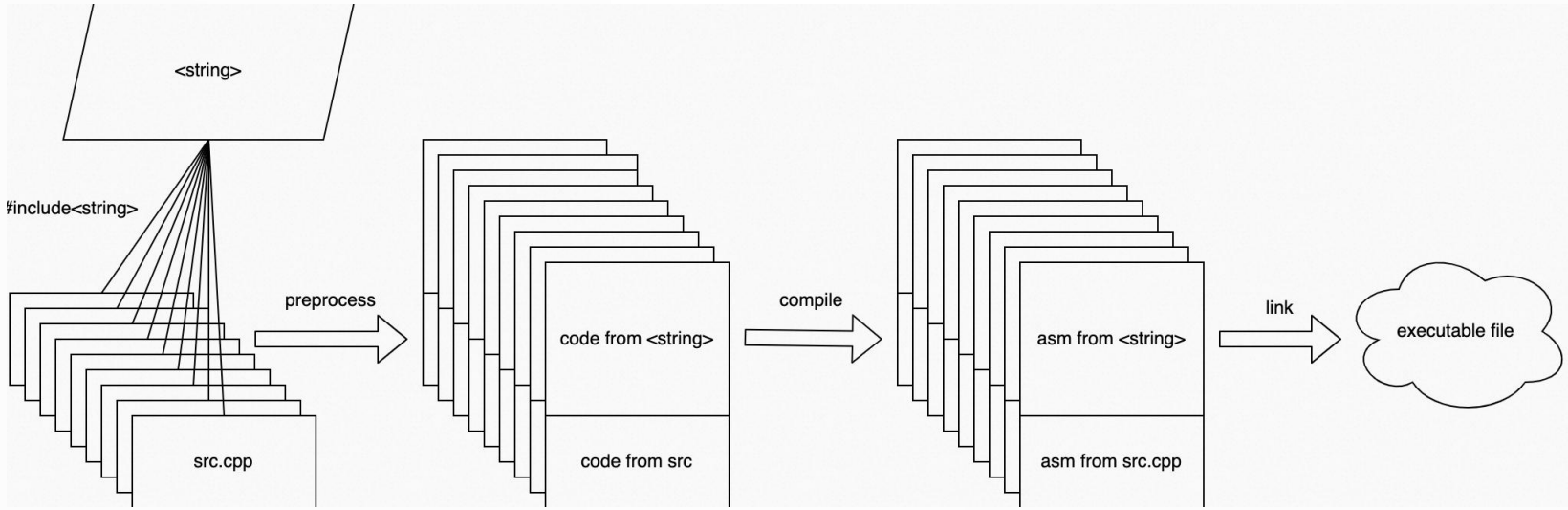
Modules

隔离性


Head File
<A> , <B> , <C>
#include
src.cpp
preprocess
code from <A>
code from <B>, affected by A
code from <C>, affected by A and B.
code from src
Module
unit A
unit B
unit C
import
src.cpp
module would't affected other by import order,because they are in different code unit.

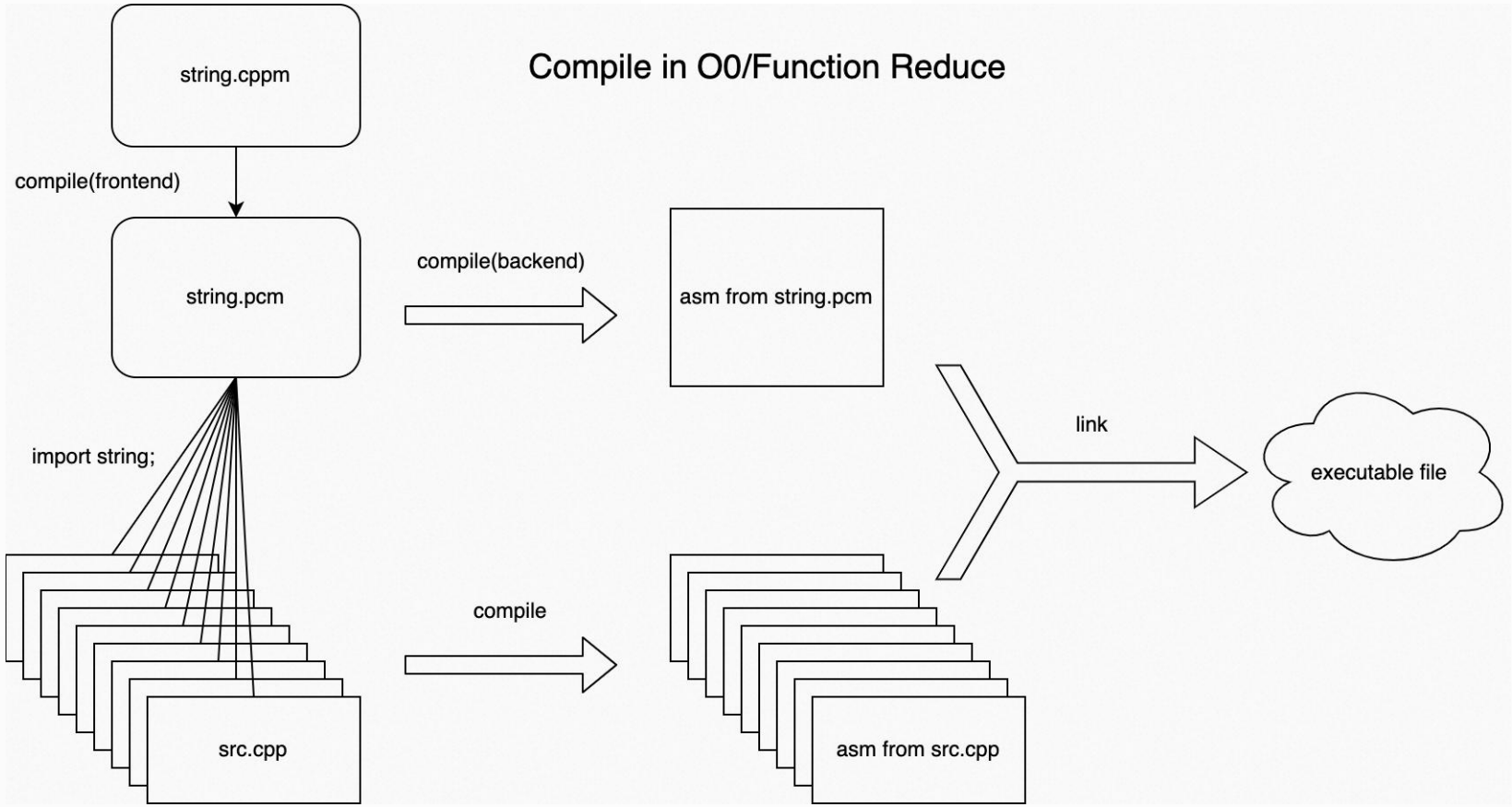
Modules

编译加速



Modules

编译加速



Modules

当前状况 – 编译器

- Clang & MSVC：Release Candidate
- GCC：缺少构建系统支持、缺少真实应用实践

Modules

当前状况 – 构建系统

- CMake：3.26 已支持 C++ Modules for clang & MSVC
- MSBuild：已支持
- Bazel: 有第三方支持；正式支持开发中
- Xmake：已支持

Modules

当前状况 – std module

- MSSTL：Released as experimental
- libc++: 开发中
- libstdc++：waiting for GCC

Modules

当前状况 – 应用情况

- 内部业务
  - 7M LoC
  - Modules 代码已合入
  - 可减少 40% 编译时间

Modules

对生态的影响

- Modules 的引入使得几乎所有依赖于 C++ 静态分析的工具都需要重新适配：
  - 构建系统
  - clangd
  - ccache
  - distcc
  - ...

Modules

对生态的影响

- 同时 Modules 的引入也让 C++ 标准委员会工具链小组看到让 C++ 的包管理系统、分发系统与其他工具之间相互接口统一的可能性。
- 与有着明确规定的 C++ 语言标准不同，C++ 工具间的互交互性是无明确规定的、经验主义和约定俗称的。
- 为了解决这个问题，工具链小组提出了《 C++ Ecosystem International Standard》。
- 长期以来，工具链小组（Tooling）的周会都在讨论 Modules 相关问题。甚至有人建议不如改名为 Modules 小组。

安全

- 内存安全是 C++ 自诞生以来便存在的问题
- 22 年 11 月，美国国家安全局发布报告，将 C/C++ 列为不安全语言，在委员会内部引起了很大的讨论
- 在标准方面，从 std::unique_ptr, std::shared_ptr 到尚未进入标准的 contracts 都是委员会为了 C++ 的安全做出的努力
- 在工具链方面，编译器与其他静态分析工具以及 Sanitizer 等长久以来一直将提高内存安全作为优先级很高的任务

安全

未来可能的改进措施

- 使用 Contracts 对代码进行约束
- 使用与 C++ 完全兼容的新语言，例如 Carbon
- 增强静态分析工具能力；将静态分析工具的要求列入 C++ 生态标准中。
- 利用 Modules 的隔离性，在不重构旧代码的情况下，对新 C++ 代码启用更严格的语法规则。

```
export module my_mod [[static_analysis(inclusions{"safe"})]]'
```

总结

- 异步化和并行化是当前 C++ 世界的主要潮流；是目前显著提升 C++ 程序性能最主流的方式。
- Modules 及其引入的生态变化是 C++ 工具链关注的主要潮流；预计在未来几年内会对 C++ 生态带来非常显著的影响。
- 对高阶 C++ 程序员而言，编译期计算会是一个可以发力的方向
- 安全问题是 C++ 自诞生以来便存在且广为人知的问题；但在最近几年变得更加尖锐；C++ 世界为了改善内存安全问题已经付出了经年累月的努力并获得了显著的成效；但由于 C++ 自身的特性，内存安全将会是 C++ 长期的议题。

GOTC

THANKS

全球开源技术峰会

THE GLOBAL OPENSOURCE TECHNOLOGY CONFERENCE