在 C++26 中枚举字段的编译成本：GCC 16 上的原生反射、结构化绑定包、反射库与手写序列化器

Denys Kachalenko独立研究员，乌克兰
denys.kachalenko@gmail.com

arXiv:2611.30250v1 [cs.PL] 23 Nov 2026

摘要

C++26 新增了枚举结构体字段的标准方式，包括静态反射（P2996，带有 P3394 注解）和结构化绑定包（P1061），同时还有手写代码、X 宏和库（Boost.PFR、reflect-cpp、Glaze），这些库如今也提供 pre-C++26 和 C++26 路径。我们在结构体序列化器中对每种方式计时，确保输出逐字节相同；测试平台为 GCC 16.2.0，分别测量前端和 -O2 阶段，并以随机交错的轮次进行测试，同时通过代码级检查验证每一项单因素比较。在前端，采用 P2996 自身多个示例中所用形式的内联原生枚举——对 define_static_array 的 std::span 执行扩展语句——其每个含 16 个标量字段的结构体的成本与 Boost.PFR 大致相当，是 P1061 参数包的 7.96× 倍；而 P1061 参数包本身与手写代码的成本相近。对常量数组执行相同循环的成本只有 0.47× 倍：span 形式的成本约有一半来自 span。在异构类型图和 -O2 下，与 P1061 的比率会缩小。在前端编译整个翻译单元时，库带来的开销超过机制本身：200 个结构体使用内联原生枚举耗时 0.59 秒，使用 Boost.PFR 耗时 0.56 秒，而使用 reflect-cpp 和 Glaze 的 C++26 路径则耗时 3.86–4.13 秒；它们的访问器还会计算字段名。对于每种库，16 个字段时，C++26 路径的单类型成本都低于 pre-C++26 路径；但字段数达到 128 时，Glaze 的收益会反转（固定版本 v9.1.0）；Boost.PFR 的 C++17 路径则会随着类型数量近乎二次增长。多个手写序列化器共享一个重载名称时，其成本会随着类型数量超线性增长。仅启用 -freflection 标志，就会使不使用反射的代码在 GCC 16.2.0 上每种类型的成本最高增加 1.264× 倍，在 16.1.0 上最高增加 2.578× 倍，因此不同机制之间的比较取决于具体的小版本。span 形式的部分成本会随着结构体数量增加；我们将此归因于 GCC 以二次时间注册块作用域静态变量（报告为 GCC PR128928）。

1 引言

序列化离不开样板代码：对每种类型，都要编写遍历其每个字段的代码。C++ 通过预处理器（X 宏）、基于聚合初始化探测的库“反射”、结构化绑定，以及对编译器生成的函数名进行语法解析（Boost.PFR、reflect-cpp，Glaze），如今又有了 C++26 语言特性：带有注解（P3394）的静态反射（P2996）、引入包的结构化绑定（P1061），以及展开语句（P1306），它们可以遍历反射成员，也可以直接解构结构体。对于实践者来说，问题很具体：这些特性各自会增加多少编译成本，又能节省多少源代码？

目前对原生 P2996 编译时间的测量才刚刚起步。Romeo [42] 测量了单个翻译单元中仅使用原生反射时的耗时；mp-units [39] 在仓库内进行了 A/B 测试，作者认为效果的方向可信，但幅度不确定；simdjson 报告称，在 clang-p2996 上启用反射后，整个构建过程慢了 6% [28]；而字段反射库 [31] 在 README 中列出了其 `std::meta` 后端的每类型成本，并与 Boost.PFR 的数据并列，采用三次测试中的最佳结果。截至 2026年11月，我们没有找到已发表的测量数据，能在同一序列化任务中比较原生反射与各库的两种路径、P1061，以及手写和宏基线（§5）。一家工具供应商的博客 [47] 称编译时影响是“最大的未解问题”。

研究问题。

- **RQ1——机制**。对于每种类型，使用 P2996、P1061、库的回调 API 以及手写方式进行字段枚举的编译成本分别是多少？P3394 注解查询又会增加多少成本？
- **RQ2——库后端**。Boost.PFR、reflect-cpp 和 Glaze 都可以从同一份源代码构建为预 C++26 路径或 C++26 路径。切换路径是否会改变其编译成本？这种变化如何随每个结构体的字段数而变化？
- **RQ3——按类型划分的序列化器惯用法**。哪些有文档记载的定制惯用法会使手写序列化的增长速度随着类型数量增加而快于线性增长？原因是什么？
- **RQ4 — 结果取决于什么**。固定开销有多大（<元数据>、库头文件以及 -freflection 标志本身）？在 -O2 优化级别下、使用更丰富的语料库和异构类型

图时，结果会如何变化？不同会话之间，以及 GCC 16.1.0 与 16.2.0 之间又有何差异？

- **RQ5 — 可维护性**。每种方法需要多少每类型源代码？

贡献。

1. 对用于序列化的字段枚举进行正面对比的编译时比较：P2996、P1061、手写代码、X 宏和三个库；每个库均测试其两种路径（同一源代码的三个后端 A/B 测试）。测试使用 GCC 16.2.0，关键分支（配置，§3.1）也在相同轮次中使用 GCC 16.1.0 编译。

2. 对原生和库枚举开销进行析因分解：枚举机制（P2996 或 P1061）× 、API 形式（内联或回调）、注解查询，以及围绕同一基本操作的库机制。本文所说的原生枚举，是指全文计时的形式：对 `std::span`中的 `define_static_array`进行1展开语句（列表 1）；析因分解表明，在前端，这种形式只有内联编写时才与 Boost.PFR 相当，若置于相同的回调 API 之后则不然。补充会话对同一成员查询的四种写法进行了计时。在 GCC 16.2.0 上，span 形式约一半的前端开销来自 span 本身；我们认为这主要是 GCC 对 C++ 所规定的迭代展开语句在其上进行求值所致。对常量数组进行内联原生枚举的开销为 0。49× Boost.PFR 的回调每个_字段_的开销为 0。3. 比较十种用于每类型序列化器的定制惯用法，分别改变调用点数量和候选项数量；共用一个重载名称的序列化器，其开销会随着类型数量超线性增长。4. 用于界定测量本身的对照：一对代码完全相同、仅库标志加入解析内容不同的实现；在两个编译器版本上测量 -freflection 标志的开销；-ftime-report 的开销；带阴性对照的值级正确性预言机；以及对所有预期只在单一因素上有所不同的分支对进行代码级操纵检查。5. 开放工件：生成器、Harness、每次运行的原始数据和环境记录。

2 背景

在 C++ 中枚举字段++26。P2996 将一个实体反射^^ 为 `std::meta::info`；`std::meta::nonstatic_`数据_成员_的 返回类型的字段，而拼接 v。`[:m:]` 访问一个字段。我们的原生序列化器通过展开语句（模板 for，P1306 [24]）遍历这些字段，调用`std::define_`静态_数组 (P3491 [12])（清单 1）。这是编写原生枚举的几种方式之一；每个 S1 和 S2 的每项任务都由原生分支计时，而另一场补充会话则为另外三种方式计时（§4.2）。P3394 将值附加到声明上（`[[=RB::FTransient{}]]`），反射可以查询这些值，例如跳过某个字段。P1061 允许结构化绑定引入一个包：在模板内部，const auto& [... m]= v; 会绑定结构体中的每个字段，例如我们的结构体（所有非静态数据成员都可访问，且在同一个类中声明），因此折叠表达式会遍历它们——无需名称或注解即可枚举（P1061R10 [40] 仅允许在模板化实体中使用此类包）。展开语句也可以解构结构体本身：template for (const auto& f : v)，限制相同（没有名称、没有注解），但在模板之外也可使用；我们没有为这种形式计时。GCC 16 实现了 P2996R13 [11], 、P3394R4 [13], 、P3491R3、P1306R5 和 P1061R10；反射需要 -freflection，GCC 将其标记为实验性功能。

预 C++ 库++26 技术。库通过使用可转换为任意类型的对象探测聚合初始化，来统计聚合体的字段；通过生成的结构化绑定阶梯（每种字段数量对应一个分支）绑定这些字段；并通过解析__PRETTY_FUNCTION__ 或 std::source_location::function_name()，获取在一个从未定义的外部对象成员地址上实例化的模板的名称。**Boost.PFR** 的经典路径 [36] 采用这种方式；在 GCC 16 上使用 -std=c++26 时，它会自动切换到基于 P1061 参数包的路径。它的 P2996 引擎（BOOST_PFR_USE_CPP26_REFLECTION）不会被 for_each_field（§3.3）调用；在固定版本中，它本身也无法序列化运行时对象，因为其成员访问（detail::reference_by_index）是 consteval。**reflect-cpp** [50] 有一条 C++ 路径，并且自 v0.26.0（2026-10-18）起，还提供一条可选启用的 C++ 路径；其文档称后者的编译开销“显著更低”，但未提供测量结果 [49]。**Glaze** [5]有一个传统后端和一个 P2996 后端。在固定版本中，传统后端通过递归链式聚合初始化探测（V{Args{}..., any_t{}}）统计成员数量，通过生成的结构化绑定阶梯绑定成员，通过解析__PRETTY_FUNCTION__ 获取名称，并且每个结构体最多接受 128 个成员；Glaze 的文档将这一计数方式称为结构化绑定探测 [3]。当编译器定义 __cpp_impl_reflection 时，Glaze 会自动选择其 P2996 后端；GCC 16 在使用 -freflection 时便会如此。

按类型定制的惯用写法。许多序列化和格式化库允许用户为每种类型编写一个同名函数，并通过名称查找找到它：nlohmann::json 的 to_json 及其 NLOHMANN_DEFINE_TYPE_NON_INTRUSIVE 宏、Boost.JSON 的 tag_invoke、BOOST_DESCRIBE_STRUCT、{fmt} 的 format_as、cereal 的外部序列化，以及 Boost.Serialization 的非侵入式序列化。当 N 个用户类型位于同一个命名空间时，每次对这类名称进行解析都会考虑 N 个候选项。nlohmann::json、Boost.JSON 和 {fmt} 在各自的分派模板中进行非限定调用，

因此每种被序列化的类型只解析一次；直接调用该名称的代码则会在每个调用点解析一次。隐藏友元（nlohmann 的_INTRUSIVE 宏、BOOST_DESCRIBE_CLASS）、成员函数（cereal 的首选形式、protobuf）以及类模板特化（glz::meta、fmt::formatter）不会产生这样的候选项集合。

3 方法论

3.1 分支

每个分支都通过一个共享的值编码器写入相同的小端二进制格式，因此输出的字节完全一致；各分支的区别仅在于枚举字段的方式。

我们将 S1 和 S2 中除 reflect-cpp 和 Glaze 之外的分支称为轻型；三个 S3 分支仅在前端惯用法中运行。这两个库的四个分支，我们称为重型，在前端每个结构体需要 12.75 到 30.56 毫秒（表 4），因此计时这些分支的任务会在 N = 200 或更早时停止。在 §3.5 的任务表以及 §4.1、§4.5 和 §4.6 中，“这些库”指的就是这些重型分支；Boost.PFR 分支属于轻型。

回调分支（reflect_visit、p1061_visit）将相同的循环或折叠操作移入函数模板 ForEachField(V, Func)，并传入 [&](const auto& Field) { Enc::Any<FSelf>(Writer, Field); }，这正是 pfr 分支传给 boost::pfr::for_each_field 的 Lambda 表达式。reflect 将调用封装在 if constexpr (!IsTransient(Member)) 中，其中 IsTransient 会测试 Member 的注解_of_with_type(Member,^^RB::FTransient)。

库分支调用的是各自库的字段访问器（boost:: pfr::for_each_field、rfl::to_view(v).apply、glz:: for_each_field），而不是自己的序列化器，因此每个分支输出的字节都相同；每个分支都包含提供相应访问器所需的最精简库头文件。嵌套结构体（包括向量和可选值中的嵌套结构体）则使用该分支自身的枚举方式。其他分支用作对照：*_split（每个调用点使用一个函数，用于 -O2）、*_noflag（使用 -fno-reflection 编译）、*_noreport（不使用 -ftime-report 编译）、arm@GCC161（在相同轮次中由 GCC 16.1.0 编译的同一分支），以及十种重载矩阵变体（§4.4）。

3.2 语料库

生成器会为每个 arm 生成 N 个结构体，以及一个基准测试翻译单元，让每个结构体各序列化一次。标量：每个结构体有 M 个字段，类型从 8 种整数、浮点数和布尔类型中循环选取；每个结构体的成员列表都相同。丰富型：在此基础上加入字符串、向量和可选值。异构型：类型图中每个结构体的字段数量各不相同（服从对数正态分布，中位数为 8，在 N = 1,600 时最多为 32），并包含枚举（占字段的 10%）、嵌套结构体（14%），以及由结构体构成的向量（8%）和可选值（6%），嵌套深度最多为 6 层。它具有前缀稳定性——前 k 个类型对所有 N 都相同——因此新增类型的边际成本有明确的定义。每个基准测试文件都会序列化通过 extern 声明的对象，因此在 -O2 下，优化器无法将已知字段值折叠进生成代码。

3.3 正确性与操纵检查

值级预言机。 整数、浮点数和字符串字段的取值均各不相同且非零；布尔字段交替取值；字符串和向量非空，可选值则有空有非空。每个分支的输出都必须与手写参考实现逐字节一致；若某个序列化器以相反顺序写入字段，则必须判定其不通过，否则说明预言机本身无法发现问题。每种计时分支使用的编译选项组合都要构建相应预言机（无反射分支不使用 -freflection），且每次构建都必须报告预期的库后端配置；预言机会在 M = 16 和 M = 128 上对标量和丰富型语料库运行，并在计时所用的最大 N 值下对异构语料库运行（1,600 种类型，样本涵盖所有形状类别）。此外，预言机还会检查十种重载变体和 P3394 路径（必须跳过 Transient 成员）。CTest 测试套件会在每种库后端下，对能够处理嵌套记录的每个分支（除 xmacro_unique 外）进行测试，记录中包含容器。

操纵检查。 对于每一对旨在仅在单个因素上有所不同的分支（后端 A/B、注解查询、API 形式、机制、S3 的原生实现、Boost.PFR 的机制、使用不同名称而非重载、仅标志控制项以及分离控制项），测试框架会编译两个分支，并通过三种方式检查预期差异是否仅出现在一侧。在目标文件中，每个后端 A/B 都必须仅在一侧存在标记函数（例如，glz::detail::get_member_ref 仅存在于 Glaze 的 P2996 构建中，tie_as_tuple 仅存在于 Boost.PFR 的 C++17 路径中，rfl::internal::cpp26:: bind_to_tuple 仅存在于 reflect-cpp 的 C++26 路径中）。在预处理后的源代码中，预期差异也必须仅出现在一侧（注解查询仅出现在 reflect 中；非静态_数据_成员_of call 仅出现在 P2996 分支中，而结构化绑定包仅出现在 P1061 分支中）。旨在保持一致的一对 ——pfr/pfr_native 以及每个 -fno-reflection 控制项——必须具有相同的反汇编结果。每个分支的标志都必须传递到实际计时所用的确切编译器命令中，并且 -fno-reflection 必须移除 __cpp_impl_reflection。对于存在差异的分支对，检查并不排除其他差异：P2996 和 P1061 分支在解析 <meta> 以及展开语句方面也有所不同；S3 在定义_static_array 的调用中还使用了不同的 span 和数组形式（§4.2、§6）。之所以进行这些检查，是因为文档中称为后端开关的标志可能无法传递到实际调用的代码：在 GCC 16 上使用 -std=c++26 时，BOOST_PFR_USE_CPP26_REFLECTION=1 不会改变 boost::pfr::for_each_field 的任何内容——无论是否启用该标志，目标文件定义的函数都相同，反汇编结果也完全一致。一次同时在许多因素上存在差异的分支对（原生实现与 Boost.PFR、库与原生分支）没有单一的预期差异，因此仅通过预言机进行检查。

Ta表 1. 序列化器分支：每个分支如何枚举字段，以及每种类型所需的源代码。

Arm	字段枚举	每类型源代码
手动 / 手动_独立	手写；一个重载名称 / 每个对象一个独立名称	每个字段
xmacro / xmacro_唯一	type X 宏字段列表；一个重载名称 / 不同名称	每个字段
reflect	P2996，再对每个成员执行一次 P3394 注解查询	none
reflect_纯文本	P2996	none
reflect_访问	基于 P2996、采用 boost::pfr:: API 形式的回调 for_each_字段	none
reflect_数组 / reflect_索引 / reflect_序列（S3 仅）	P2996，同一个成员查询：展开语句 遍历常量数组（[:reflect_constant_array(...):]) / 由 make_index_ 生成的 P1061 索引包 序列（reflect-cpp 的 C++26 形式）/ 一个索引序列 辅助工具	none
p1061 / p1061_访问	P1061 参数包，通过同一回调 API 内联实现	none
pfr	Boost.PFR，在 -std=c++26 下默认使用（其 P1061 路径）	none
pfr_经典	Boost.PFR，C++17 路径（-DBOOST_PFR_USE_CPP26=0）	none
pfr_原生	Boost.PFR，使用 -DBOOST_PFR_USE_CPP26_REFLECTION=1：编译生成的代码与 pfr 相同（反汇编完全相同，§3.3）；该标志只会在解析时添加头文件（代码相同对照组，§4.2）	none
rflcpp / rflcpp_原生	reflect-cpp、C++20 path / C++26 path	none
glaze_经典 / glaze_原生	Glaze，传统后端（强制）/ P2996 后端	none

这些检查针对每个编译器各运行一次，并绑定到测量代码（测试框架模块、Include/RB 和库版本固定项的指纹）以及每个编译器的 cc1plus 哈希值：只有当某个会话中的每个编译器都针对完全相同的代码通过检查时，该会话才会启动：S1 和 S2 在两种编译器上各有一次通过检查；而仅使用 GCC 16.2.0 的 S3 则针对其自身代码完成了一次检查，该代码加入了三个分支。每次会话开始前，都会再次验证库版本固定项，并使用 GCC 16.2.0 重新运行 CTest 测试套件。

代码溯源。 本研究的代码是在作者指导下借助 AI 助手编写的（致谢）；代码已通过上述检查，并通过从归档原始数据重新生成论文中的每个数值，再将其与正文进行核对而得到验证（制品）。

3.4 测量内容

该指标是编译器自身的 -ftime-report TOTAL：在编译器内部测得的墙钟时间（GCC 16 报告墙钟时间和垃圾回收内存，而非 CPU 时间），以及其阶段和子阶段计时器与垃圾回收（GGC）内存。对于每次编译，Harness 还会记录外部墙钟时间，并从已退出的进程中读取其用户态和内核 CPU 时间、CPU 周期、峰值工作集、峰值提交量和缺页错误。Harness 直接运行 cc1plus，使用 g++driver 生成的完全相同的命令行（-###），并设置较高的进程优先级，将其限制在混合 CPU 的性能核心上。直接运行 cc1plus

很重要：在 Windows 上，高优先级驱动程序的子进程会以正常优先级运行，因此通过驱动程序启动会使计时进程失去保护。两种模式：-fsyntax-only（前端：语法解析、模板实例化、常量求值）和 -O2 -c（增加优化和代码生成；汇编代码写入空设备，因此不会运行汇编器）。每种类型的数字都不包含固定开销：它们是 N（§3.6）上的斜率，或边际成本，即 N 个结构体的成本减去 N 个 $= 0$ 的成本。

3.5 协议

一次会话会运行下面的任务。在每项任务中，每个单元都会在一次预热轮次和十一次测量轮次中各编译一次（arm × sweepvalue），并以固定种子的随机顺序安排在两个固定哨兵单元的编译之间，用于跟踪漂移。同一比率中进行比较的分支始终来自同一项任务，因此共享这些轮次。在计时之前，Harness 会记录计算机信息（CPU 和核心布局、电源计划、防病毒软件状态、每个 CPU 的负载以及最繁忙的进程）、每个编译器的哈希值和配置命令行、库版本固定项，以及测量代码的指纹，并保留一份副本；它会将会话绑定到相应的编译器、代码和协议，因此恢复会话时不会悄悄混用这些内容。运行时采用高性能电源计划。轮次不会重复：每次编译期间计算机的 CPU 负载都会与该次编译一同记录，每个估计值都是十一次测量轮次的中位数或拟合结果。在不同日期运行的两个会话会在 GCC 16.2.0 上测量完整方案；比较任务则在相同轮次中并排使用 GCC 16.2.0 和 GCC 16.1.0 编译各自的分支。A

清单 1。 分支的核心代码（来自 Include/RB 和生成器）。Enc::Any 是共享值 en 编写器；它将嵌套结构体传回 armthroughFSelf。// reflect_plain：P2996 查询、P1306 展开语句、P3491 静态数组模板<typename Type>void Write(FByteWriter& Writer, const Type& V) { constexpr auto Context = std::meta::access_context::unchecked(); 模板 f or (constexpr auto Member : std::define_static_array(std::meta::nonstatic_data_members_of(^^Type, Context))) { Enc::Any<FSelf>(Writer, V.[:Member:]); } } // reflect_数组 (S3)：对常量数组执行相同循环，而不是使用 std::span // 模板 for (constexpr auto Member : //[:std::meta::reflect_constant_array(std::meta::nonstatic_data_members_of(^^Type, Context)):] // p1061：结构化绑定包和折叠表达式 模板 <typename Type>void Write(FByteWriter& Writer, const Type& V) { const auto& ［... Members］= V; (Enc::Any<FSelf>(Writer, Members), ...); } // 手动_唯一：生成的每种类型各有一个非模板函数 inline void WriteManualS0(RB::FByteWriter& Writer, const S0& V) { RB::Enc::Value(Writer, V.M0); // ... 每个字段对应一条语句 }

第三次短会话（S3，2026-11-21）按照相同协议，在 GCC 16.2.0 上运行了一个补充任务 FrontendIdiom；在审查前两次会话后加入该任务，是为了测量其他原生实现形式的耗时，其预检也像检查其他任务一样，检查了三个新增分支。会话开始前，其中七个分支中的六个（除后来添加的数组形式外）各自非正式地编译了一次；加入数组形式后，又对整个任务运行了一轮冒烟测试；这里不采用这两项结果。其分析规则在会话开始前已确定：仅在任务内部比较各实现形式；只有通过数组/s- pan 对，才能将影响归因于展开语句的降低；与 S1 共享的分支则仅与 S1 进行描述性比较。这些共享分支在每个 N 下分配的 GGC 内存都与 S1 相同。

S1 和 S2 各运行了 8,484 次编译，均未失败。在测量期间，计算机的总 CPU 负载（覆盖全部 20 个逻辑 CPU，其中单线程编译器本身最多占 5%）在 S1 和 S2 中的中位数分别为 8.0% 和 7.3%；负载超过 15% 的编译分别占 5.1% 和 2.0%。在计算单元格中位数前剔除这些编译后，S1 中通过线性检查且增幅至少达到十个计时器步长的 96 个逐臂斜率里，有 90 个变化不超过 1%（在每个个任务中，ReportOverhead 除外；它按外部墙钟时间进行比较），最大变化为 2.1%；S2 的 97 个斜率变化均不超过 0.6%。对于耗时超过 2 秒的编译，S1 和 S2 中分别有 99.3% 和 99.2% 的案例，其 TOTAL 与进程 CPU 时间（用户态+ 内核态）相差不超过 3%，因此这些编译几乎没有花时间等待 CPU；每个任务中的哨兵单元格都保持在其中位数的 4.7%（S1）和 5.2%（S2）以内。S3 运行了 444 次编译，均未失败；其中位负载为 8.0%（负载超过 15% 的编译占 2.9%）。剔除其负载超过 15% 的编译后，各斜率的变化均不超过 0.3%。在耗时超过 2 秒的 111 次编译中，TOTAL 有 95.5% 与 CPU 时间相差不超过 3%，这一比例低于 S1 和 S2；其他编译在任何单元格中都不超过一次，因此单元格中位数会将其影响吸收。其哨兵值保持在其中位数的 4.8% 以内。

3.6 统计

每个单元格的估计值是所有测量轮次的中位数。对于通过下文线性检查的曲线，每类型成本是单元格中位数对 N（或 M、K、C）进行普通最小二乘回归所得的斜率。不确定性通过对轮次进行自助法估算（4,000 次重抽样）：抽取轮次时采用

表 2。 会话的任务（§3.5）。

Job	Mode	语料库	扫描	Arms
FrontendN	前端	标量，M = 16	N ∈ {0, 500, 1,000, 2,000, 4,000}	所有轻型武器，包括手动、xmacro 以及 pfr_原生
FrontendHeavy	前端	标量，M = 16	N ∈ {0, 25, 50, 100, 200}	reflect-cpp 和 Glaze，两条路径， + reflect_普通，p1061_visit，pfr
FrontendM	前端	标量，N = 50	M ∈ {0, 8, 16, 32, 64, 128}	轻型武器（手动、xmacro 和 pfr_原生、 + 重型武器
RichN	前端	丰富型，M = 16	N ∈ {0, 25, 50, 100, 200}	S1 和 S2 的每个串行器臂
HeteroN	前端	异构型	N ∈ {0, 200, 400, 800, 1,600}	轻型武器（xmacro 除外）_独特，无法嵌套
HeteroHeavy	前端	异构型	N ∈ {0, 25, 50, 100, 200}	重型武器 + reflect_普通，p1061_访问，pfr
FullN	-O2	标量，M = 16	N ∈ {0, 50, 100, 200, 400}	轻型武器，xmacro 和 pfr 除外_原生， + 分离控制
FullHeavy	-O2	标量，M = 16	N ∈ {0, 25, 50, 100}	重型武器 + reflect_普通，p1061_visit，pfr
OverloadN	前端	标量，M = 16	K =C = N ∈ {0, 250, 500, 1,000}	十种惯用法
OverloadC	前端	标量，M = 16	C ∈ {0, 250, 500, 1,000, 2,000}，K = 2,000（每种类型也各调用一次）	自由函数、序列化模板，分派模板，不同名称
OverloadK	前端	标量，M = 16	K ∈ {500, 1,000, 2,000, 4,000} 在 C = 500	自由函数，序列化模板，不同名称
CompilerN, CompilerFlag	前端	标量, M = 16	N ∈ {0, 500, 1,000, 2,000}	GCC 16.2.0 和 16.1.0 上的关键分支，成对的；带有和不带有 CompilerFlag
CompilerHeavy	前端	标量，M = 16	N ∈ {0, 50, 100, 200}	不带 -freflection Glaze 和 reflect-cpp 的两种路径 + reflect_plain，在两种编译器上，成对的
FlagCost、FlagCostHeavy	前端	标量，M = 16	N ∈ {0, 500, 1,000, 2,000} / {0, 50, 100, 200}	启用和未启用 -freflection
FrontendIdiom（仅限 S3）	前端	标量，M = 16	N ∈ {0, 500, 1,000, 2,000, 4,000}	reflect_普通，reflect_数组，reflect_索引，reflect_序列，P1061、pfr、手动_唯一
报告开销	前端	标量，M = 16	N ∈ {0, 1,000, 2,000}	reflect_普通、pfr、手动_有和没有时各自唯一 -ftime-report（外部墙钟时间）

有放回抽样，并在每次抽样中重新计算所有统计量——单元格中位数、斜率、比值、占比和交叉点——从而保留各分支在同一轮次内的配对关系。由于 -ftime-report 以 10 ms 为步长输出墙钟时间（GCC 16 可以将同一报告以更高精度写入 SARIF 输出，但我们没有采用这种方式），每个重抽样值都会加上一个步长范围内的均匀抖动（平滑自助法），避免近乎恒定的单元格产生宽度为零的区间；GCC 省略的计时器行（低于约 5 ms）按零计算。点估计使用未经平滑的中位数，因此估计值可能略微超出平滑后的区间；文中引用的两个数值都出现了这种情况，且偏差均小于 0.01。区间是单次会话内的 95%百分位区间。

对于非线性曲线，我们报告各点的数值或拟合出的二次曲线，而不报告斜率。当曲线整个扫描的上半段斜率与下半段斜率相差超过 1.25 倍时，我们将其视为非线性。

并且该比率的 95% 区间不包含 1；或者其线性拟合的 $R^2 < 0.98$，且与某个数据点的偏差超过三个计时器步长。如果曲线在任一半段的增幅不足十个计时器步长，则跳过半段检验，因为计时器无法显示曲率；R^2 检验仍然适用。通过检验的曲线仍可能弯曲，因此斜率是对整个任务扫描范围的平均值，并取决于该扫描范围：reflect_plain 在 $N \le 4{,}000$（FrontendN）时，每个结构体耗时 1.72 毫秒，在 $N \le 200$（FrontendHeavy）时耗时 1.39 毫秒；pfr 分别为 1.66 和 1.39 毫秒。这进一步说明，比率只能在同一任务内计算。

除非另有说明，§4 中的所有测量数据均来自会话 S1；§4.2 的原生实现形式来自 S3（任务 FrontendIdiom），而 §4.3 的 Boost.PFR 探针则是单独进行的一次运行。会话 S2 在另一天使用相同的编译器、代码和协议重复了完整计划。两个会话无法估算会话间方差，因此我们不会合并不同会话的区间；我们

不跨会话合并区间；§4.6 则按 arm 对比各会话，计算 S2 中每种类型斜率与其 S1 值的比率，并给出主要比率的合并点估计。之后通过重采样较少轮次来检查轮次数量：每个单元格采用五轮而非十一轮时，FrontendN 斜率的 95% 区间宽度从估计值的 1.5% 扩大到 2.1%（reflect_plain），从 2.2% 扩大到 3.3%（p1061），以及从 1.4% 扩大到 2.0%（pfr）。

3.7 工具链、库、计算机

基于发布压缩包构建的 GCC 16.2.0（已根据发布的校验和验证 SHA-512，并验证签名），采用与用于比较的 GCC 16.1.0 相同的 configure 选项；两者均未采用自助法构建，且由相同的主机编译器（MSYS2 GCC 15.2.0）构建，安装布局也相同（前缀中包含 MinGW 头文件，编译器可执行文件已剥除调试信息），`--enable-checking=发布`，标志 `-std=c++26 -freflection`。两款编译器的配置均使用 `--with-tune=原生`，驱动程序会在`-mtune=alderlake` 的选项中解析此设置。本次使用的库均固定到当时（2026年11月17日）的最新发布版本，并以经过 sha256 校验的归档文件获取：Boost.PFR 来自 Boost 1.92.0 (401385c) [36]，reflect-cppv0.27.1 (e0ee3f6) [50], Glazev9.1.0 (43b175e) [5]。计算机：英特尔酷睿 i5-14600KF（6 个性能核心和能效核心；Harness 和编译器运行于性能核心的 12 个逻辑 CPU 上），32 GB，Windows 11（build 26300），已开启 Microsoft Defender 实时保护，MinGW-w64 UCRT。

4 结果

4.1 固定成本（RQ4）

包含 <meta> 以及原生分支的头文件，会使一个已包含 <string>、<vector> 和 <optional> 的翻译单元额外增加 30 ms[23, 37] 的开销（reflect_plain 与 p1061 在 N = 0 时的对照，相同任务）。库的头文件（能提供各访问器的最精简头文件，§3.1）开销要高得多。在 N = 0 时，相较于 Boost.PFR，reflect-cpp 在其 C++20 路径上增加 620 ms，在其 C++26 路径上增加 550 ms；Glaze 使用传统后端时增加 1260 ms，使用 P2996 后端时增加 1270 ms。按照设计，以下所有每类型数据均不包含这些固定开销。新增头文件是否也会改变每类型成本，由代码相同对照组测量（§4.2）。

The-freflection 标志。 对于不使用反射的代码，启用该标志会使每种类型的前端开销增加：手写代码为 1.188 倍，P1061 为 1.239 倍，Boost.PFR 为 1.082 倍；Glaze 和 reflect-cpp 的预 C++26路径分别为 1.039 倍和 1.017 倍。它还会使固定开销增加一到三个 10 毫秒计时器步长（Boost.PFR 为 10 毫秒 [1, 16],，Glaze 为 30 毫秒[22, 51]）。启用或不启用该标志，编译后的代码都完全相同（§3.3）。在 GCC 16.1.0 上，该标志的开销大得多（§4.6）。GCC 16.0 在 <bits/stdc++.h> 上启用该标志的开销为 19%（PR124925 [33]）；GCC 16.2.0 除其他反射改动外，还包含重写了在另一种情况下导致性能降低 40 倍的检查逻辑（PR 125179 [29]）。本研究中的每种序列化器实现方案都在启用该标志的情况下编译（只有 *_noflag 对照组未启用），因此下文的比较中，原生实现方案不会受到仅由该标志造成的劣势；而项目若为了使用反射而启用该标志，也必须承担它对不使用反射的代码造成的开销。

表 3. 每个结构体的前端开销（标量语料库，M = 16，FrontendN，N 最大为 4,000；sessionS1）。

Arm	毫秒/结构体 95% 置信区间	
手动_唯一	0.23	[0.23，0.23]
xmacro_唯一	0.24	[0.24，0.25]
P1061	0.22	[0.21，0.22]
p1061_访问	0.45	[0.45，0.46]
reflect_普通	1.72	[1.70，1.73]
reflect_访问	2.01	[2.00，2.02]
reflect	2.36	[2.32，2.37]
pfr	1.66	[1.65，1.67]
pfr_原生（与……代码相同	1.77	[1.75，1.78]
pfr)		

测量工具。 使用 -ftime-report 编译会使外部墙钟时间的每结构体斜率提高：原生分支提高 1.028×，Boost.PFR 提高 1.041×，手写代码提高 1.123× [1.094；在 N = 2,000 时对整个翻译单元进行测量，三者分别为 1.031、1.042 和 1.086。相对于自身成本而言，手写代码承担的开销最大，因此以它为基准的比率在使用报告时会更低：在外部墙钟时间方面，使用报告时，原生分支每个结构体的成本是手写代码的 6.92× 倍，不使用报告时为 7.56× 倍；而其与 Boost.PFR 的比率无论是否使用报告都相同（1.00 和 1.01）。

4.2 这些机制的每类型成本（RQ1）

Boost.PFR 的 C++17path（pfr_classic）没有对应行：其成本并非随 N 呈线性增长（§4.3）。手动方式和 xmacro 也一样，它们共用一个重载名称（§4.4）。其他行均通过了 §3.6 的线性检查。不过，P2996 分支（全部采用 span 形式；下文的 S3 其他形式没有上升）、p1061_visit 和 Boost.PFR 的每个结构体成本都会随 N 上升，因此它们的斜率是扫描范围内的平均值。内联 P1061 和手写代码的成本保持平稳：前 500 个和最后 2,000 个结构体的每个结构体耗时分别为 0.20 和 0.21 ms，以及 0.22 和 0.23 ms。原生分支的每个结构体边际成本，即（N – 时的成本 = 0N 时的成本）/ N，在 N = 500 时为 1.44 ms，在 N = 4,000 时为 1.71 ms。Boost.PFR 的对应值为 1.44 和 1.65 ms。上升并非 P2996 独有：原生分支的上半段斜率为 1.14× [1.12，是其下半段斜率 1.17] 的两倍；p1061_visit 的斜率为 1.23×。在 span 形式下，整个翻译单元的并发名称查找计时器从 N = 1,000 时的 0.12 秒增长到 4,000 时的 0.98 秒。这与原生分支超出恒定每结构体成本的增幅大致相当。GCC 16.2.0 会为常量迭代展开语句声明范围、起始迭代器和每个元素的迭代器

变量（每次展开有 2 + 个，覆盖 n 个元素）。它会通过对整个翻译单元中的一个列表进行线性扫描，注册每个块作用域静态变量（在 cp/decl.cc 中确定_local_discriminator）。因此，每次展开都要为此前编译的所有这些变量付出成本。即使没有任何反射，普通块作用域静态变量也会呈现相同的二次增长；从 GCC 11.5 到当前开发分支的每个 GCC 版本都是如此。这是 GCC 的编译时问题，而非 P2996 的特性；此问题已作为 PR128928 [22] 报告（复现程序见 Repro/GccLocalStatics）。在 S2 中，表 3 的斜率相较于 S1 的值分别为 1.000×（manual_unique）、1.009×（p1061）、1.002×（pfr）、0.990×（reflect_plain）和 0.995×（reflect）。

机制 ×API 形式。在这里，与 Fron- tendIdiom 以外的所有任务一样，P2996 枚举指的是列表 1 中的 span 形式；仅其表述方式就会使成本相差约两倍（见下文 S3）。当序列化器以内联方式编写时，P2996 枚举的每个结构体成本是 P1061 参数包的 7.96× [7.83、8.04] 倍（前 500 个结构体为 7.20× 倍，最后 2,000 个为 8.47× 倍）。通过回调 API 时，比率为 4.44× [4.40、4.48]。回调形式本身在 P2996 下的成本为 1.17×，在 P1061 下为 2.10× 。Boost.PFR 基于相同回调形式背后的同一 P1061 基元，其自身机制使其成本达到 p1061_visit 的 3.66× [3.63、3.70] 倍：在 -std=c++26 下，它的 for_each_field 会绑定参数包，构建一个与字段数量相等的索引序列，再次绑定参数包，并针对每个字段调用一个分派器（for_each_field_impl_apply）。该分派器接收作为类型的字段索引（size_t_<I>），并首先通过替换失败并非错误（SFINAE）测试回调是否也接受该索引。

原生枚举与 Boost.PFR 的对比。以内联方式采用 span 形式编写时，原生枚举的每个结构体成本与 Boost.PFR 的 for_each_field 大致相当。这种持平描述的是采用 span 形式的内联原生序列化器与 Boost.PFR访问器之间的选择，而不是两种枚举机制之间的比较。斜率比率为 1.04× [1.02, 1.04]。边际成本的比率取决于 N，因此哪种分支更便宜也取决于 N：当 N 为 = 1,000 时是 0.95× [0.94, 0.97]，在 2,000 时是 0.99× [0.98, 1.01]，在 4,000 时是 1.03× [1.02, 1.04]。在 CompilerN 任务（N 为 ≤ 2,000）中，斜率比率为 0.99×，在丰富语料库上为 0.96×，在异构语料库上为 0.99×（1.07× [1.05, 1.09]，N 为 ≤ 200，HeteroHeavy；§4.6）。所有这些差异都与下文代码相同对的差异（1.03–1.06×）相当或更小。在 API 形式一致的情况下，通过相同回调（reflect_visit）实现的原生枚举，每个结构体的成本是 Boost.PFR 的 1.21× [1.20 倍（S2：1.21），在丰富语料库上为 1.14×，在异构图上为 1.08× 。在每个语料库上，这都超过了代码相同对之间的差异，不过原生分支还会解析 <meta>，而该代码相同对表明这会带来几个百分点的成本。1.21 是上述两种分解结果的商。通过回调实现时，P2996 枚举的成本是 P1061 的 4.44× 倍，而 Boost.PFR 的机制与相同原语相比成本为 3.66× 倍。原生枚举通过内联编写与 Boost.PFR 持平，从而避免了回调形式带来的 1.17 倍开销。以内联方式对常量数组进行操作而不是使用 span 时，其成本约为 Boost.PFR 的一半，每个字段的 for_each_field（见下文），同样没有保持 API 形式一致。

原生实现（S3）。上述原生实现方案均采用同一种形式进行枚举，这种形式也用于 P2996R13 自身的多个示例 [11]（例如其通用格式化器）：对 std::span 执行展开语句，而 std::span 由 define_static_array 返回。C++26 将其实现为迭代展开语句，通过 std::span 的迭代器逐个定义元素，GCC 16.2.0 也以这种方式对其求值。编译器对遍历该范围的循环进行常量求值以统计元素数量，然后针对每个字段解析 begin +i 及其在 std::span 类类型迭代器上的解引用，推导 auto，并对静态 constexpr 迭代器和成员进行常量初始化。对于数组，C++26 则通过解构展开。会话 S3 在一个任务中，将相同的成员查询分别计时为另外三种形式，并与 span 形式进行比较（FrontendIdiom，N ≤ 4,000；图 2）。每个结构体中，对常量数组执行相同展开语句（[:reflect_constant_array(...):]）的耗时为 0.82 ms，而对 thespan 的耗时为 1.73 ms；边际成本分别为 0.47× [0.47 和 0.48]（在 N = 1,000 时为 0.53× ，在 4,000 时为 0.48× ，因为只有 span 形式的耗时随规模增长）。计入固定成本后，在 N = 1,000 时，采用数组形式的整个翻译单元前端耗时为 span 形式的 0.61×；在 4,000 时则为 0.50× 。索引序列辅助工具耗时为 0.93 ms（0.54×）；reflect-cpp 的形式则耗时 2.02 ms（1.17×）：它通过元组协议绑定一个 constexpr P1061 索引包，而 C++26 会将其提供为 std::integer_sequence（P1789R3）。该形式的重载决议计时器占其耗时的 54%，这与每个元素都通过实参依赖查找解析一次 get<j> 相符。

因此，span 形式每个结构体的成本约有一半来自 span 本身，而不是成员查询。这一对比还排除了 static_array 在 reflect_constant_array 周围执行的调用，包括 type_of、is_array_type、extract、extent 和 span 构造函数；我们没有单独计时这些调用。它们每个结构体只运行一次，而迭代展开的逐元素步骤每个字段运行一次，因此我们主要将差异归因于迭代展开。没有证据表明展开语句本身成本高：不包含展开语句的两种形式表现方向相反，而四种形式中成本最低的本身就是一种展开语句，即对数组进行展开的形式（0.88× the helper）。采用数组形式时，内联原生枚举的成本为每个结构体的每个字段 3.77×inline P1061 和 0.49× Boost.PFR 的回调（S3 没有回调形式的数组分支）。它比手写代码高出 0.59 毫秒，并且不会随 N 增长（上半段/下半段斜率为 0.99，而 span 整体为 1.15）；另外两种形式也一样（P1061 索引包为 0.97，索引序列辅助工具为 0.98）。同样在 FrontendN 中运行的分支重现了它们在 S1 中的数值：span 形式为 1.73，而非 1.72 毫秒；Boost.PFR 为 1.68，而非 1.66 毫秒；P1061 为 0.22，而非 0.22 毫秒；手写代码为 0.23，而非 0.23 毫秒；原生 /



图 1. 标量语料库中每个结构体的前端开销，M = 16，会话 S1，对数刻度。P2996 分支使用清单 1 的 span 形式。每个点表示单元格中位数（测量了十一轮）相对于 N 的最小二乘斜率；误差线表示 95% 自助法区间，且大多比点本身还窄。(a) FrontendN 作业，N = 0–4,000：表 3 中的各分支，均通过 §3.6 的线性检查；手动、`xmacro` 和 Boost.PFR 的 C++17 路径不随 N 呈线性变化，因此未列出。(b) FrontendHeavy 作业，N = 0–200：reflect-cpp 和 Glaze 的两条路径，以及原生分支、Boost.PFR 和 `p1061_visit` 作为参考。参考分支在 (b) 中的每个结构体成本低于 (a)，因为其每个结构体的成本会随翻译单元中已有类型数量增加；因此，比率仅在同一面板内计算。

P1061。S3 是一次单独的会话，因此其区间低估了日间变化；在 S1 和 S2 中都计时的斜率其中位数为 0.7%，最高为 8.8%（§4.6），远低于 span 形式与数组形式之间两倍的差距。

S3 仅在前端对 M = 16处的标量语料库进行了计时；本文中的所有其他原生数值均采用 span 形式。

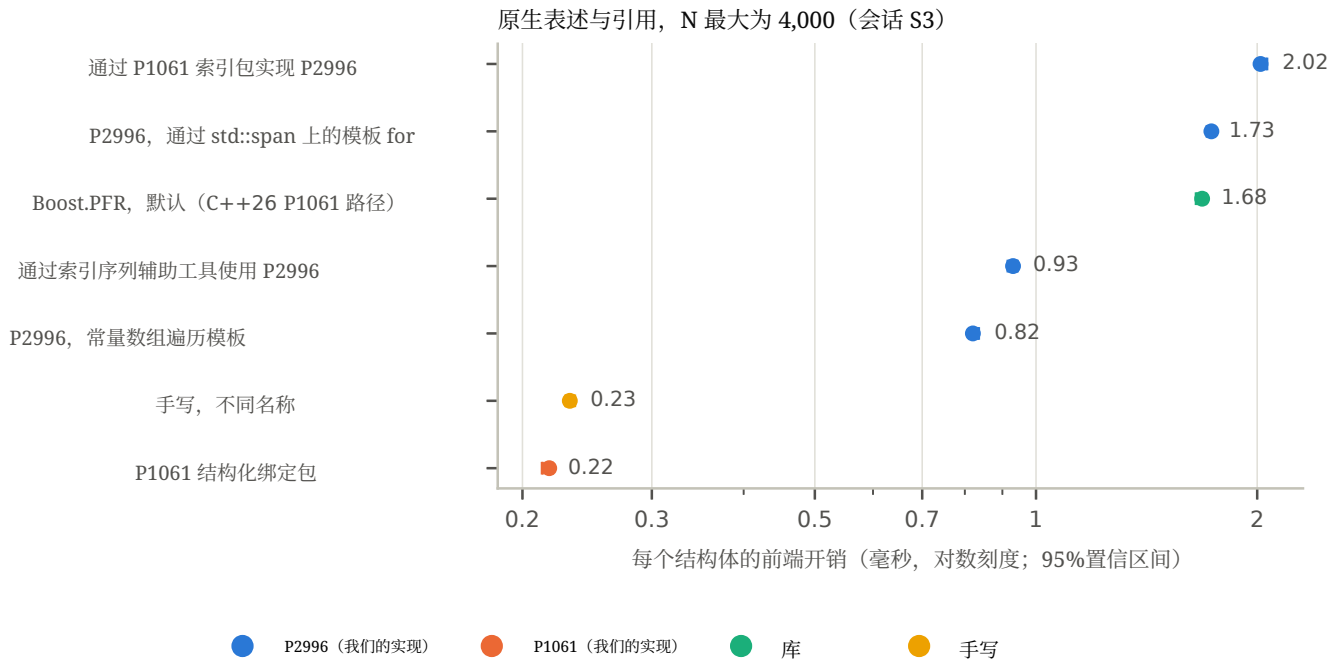


图 2. 四种原生实现形式对同一成员查询及三种参照分支的每个结构体前端开销，标量语料库，M = 16，会话 S3（任务 FrontendIdiom，N = 0–4,000），对数刻度。每个点表示以 N 为自变量、对单元格中位数（测量了十一个轮次）进行最小二乘拟合所得的斜率；误差线表示 95% 自助法区间。span 形式是其他每项任务的原生分支。

注解查询。 在每个成员上查询__带有_类型的注解，会使原生序列化器在每个包含 16 个标量字段的结构体上的成本增加至 1.37× [1.36、1.38]。在此语料库中，没有成员带有注解，因此该查询纯属开销，换来的只是能够跳过带注解的字段。

时间花在哪里。 GCC 16 有三类计时器。阶段（解析阶段、语言延迟阶段、优化与生成阶段）划分了整个运行过程。模板实例化和常量表达式求值是彼此互斥的栈计时器。 |名称查找和 |重载决议是并行计时器：它们会在查找或调用触发的推导、替换和实例化过程中持续计时，因此与其他计时器均有重叠。原生分支每个结构体耗时中，常量求值占 43%，模板实例化占 46%，两者合计占了大部分。Boost.PFR 对应的比例为 1% 和 84%，P1061 则为 1% 和 79%。这两种原生实现形式的占比都包含展开语句逐个元素处理的耗时，而不仅仅是反射查询。在 S3 中，span 形式的 1.73 毫秒耗时中，46% 用于模板实例化，43% 用于常量求值；数组形式的 0.82 毫秒耗时中，这两项分别占 35% 和 53%。因此，两种形式的耗时差异有一半以上来自模板实例化。我们没有单独测量 P2996 操作的耗时。

内存与耗时相比，会以不同方式排列各分支。按每个结构体计算，原生分支的编译器分配了 266 KB 的 GGC 内存，峰值工作集增加了 235 KB；Boost.PFR 则是在每个结构体的耗时大致相同的情况下，分别为 426 和 408 KB；P1061 为 60 和 50 KB；手写代码为 73 和 60 KB。注解查询使原生分支的这两项数值升至 382 和 333 KB。每个单元格在每一轮的 GGC 内存都相同，峰值工作集的中位数也几乎如此：两者按每个结构体计算的区间宽度都小于 0.1 KB。

手写代码为每种类型使用不同名称，是成本最低的三种分支之一，另外两种是 P1061 参数包和使用不同名称的 X 宏（表3）。每个结构体中，P1061 的成本为 0。94× [0.92，0。94] 而 X 宏为 1。06× [1.04，1。06]，与手写分支相当。我们没有测量这两个分支中测量工具的开销；在手写分支中，它使每个结构体的墙钟时间增加了 1。123×（§4.1），增幅大于这些差异，因此我们认为这三个分支的表现相当，而不对其排序。在标量语料库中，原生枚举每个结构体的成本是手写代码的 7。46× 倍；若不使用`-ftime-report`（§4.1），这一比率还会高出约 9%；使用注解查询时则为 10。22×，但我们没有测量其测量工具的开销。

代码相同对照组。pfr_原生实现编译出的代码与 pfr（§3.3）相同，每个结构体分配的 GGC 内存也相同（426 KB）；该标志仅将 Boost.PFR 切换为基于反射的定义，这些定义对于_每个_字段都不会实例化，且位于包含 <meta>的头文件中。然而，在前 500 个结构体中，其每结构体耗时为 pfr 的 1.06× [1.05 倍；在最后 2,000 个结构体中，则为 1.07] 倍（分别为 1.04× 和 1.07×）；合并两个会话计算的结果也相同（1.06×）。在异构语料库中

表 4. 同源后端 A/B 测试：每结构体前端成本（毫秒），先前路径 / C++26 路径（标量语料库，M= 16）。reflect-cpp 与 Glaze 行：N = 0–200 上的斜率（FrontendHeavy）；Boost.PFR 行：在所述 N 处每个结构体的边际成本（FrontendN）[a]。

库	先前路径	C++26 路径	比率	95% 置信区间
reflect-cpp	30.56	14.66	2.08	[2.06，2.10]
Glaze	25.46	12.75	2.00	[1.96, 2.01]
Boost.PFR, N = 500	6.32	1.44	4.4	[4.3、4.5]
Boost.PFR, N = 4,000	21.07	1.65	12.8	[12.7、12.9]

a Boost.PFR 的 C++17 path 并非关于 N 呈线性，因此没有每结构体斜率。其各行给出了指定 N 下每个结构体的边际成本，即对于两条路径，(N − 时的成本 = 0N 时的成本) / N，以及它们的比率。

该比率为 1.03× [1.01，即 1.04]；在丰富语料库中，N 为 ≤ 200 时，结果尚未确定（ 0.97× [0.95，1.02] ）。因此，翻译单元解析的内容会改变未改动代码的每类型成本，幅度可达几个百分点，而不仅仅影响固定开销。原生实现方案会解析<meta>；P1061、手写和 Boost.PFR 实现方案则不会。因此，它们之间几个百分点的差异不能归因于枚举机制。我们没有测试单独的 <meta> 是否会造成这种影响。

4.3 库后端（RQ2）

Boost.PFR 的 C++17 路径随类型数量呈二次增长。 从 N = 2,000 到 4,000，其边际前端开销增至 3.7 倍（C++26路径：2.0 倍），在 N = 4,000 时达到 84 秒，其中重载决议计时器占 87%；GGC 内存呈线性增长（每个结构体 1262 KB），而在 N =4,000 时，整个翻译单元的工作集峰值达到 5.0 GB（C++26 路径：1.6GB）。在丰富语料库上，扫描范围截止于 N = 200，曲线仍接近直线（从 N= 100 到 200，增幅为 ×2.2）；在异构语料库上，曲线在扫描范围内就已开始弯曲（从 N = 800 到 1,600，增幅为 ×2.6；上半区间的斜率为 1.59× [1.55，是下半区间斜率的 1.64] 倍，因此根据 §3.6 并非线性）。因此，对于 reflect 许多类型的翻译单元，二次项不可忽视。另一个独立探测（Harness/ pfrprobe.py；在性能核心上逐个编译，取三次运行的中位数）对其进行了定位。

- 既不是 *GCC 16*，也不是这个标志。在 GCC 15.2.0 上使用 `-std=c++20` 时，耗时增长趋势相同（N 为 = 1,000、2,000 和 4,000 时分别为 7、21 和 78 秒）；在 GCC 16.2.0 上使用 `-std=c++20` 和 `-std=c++26` 时也一样，无论是否启用 -freflection。重载决议计时器还涵盖了决议过程中触发的推导和替换，在所有情况下，N 为 = 4,000 时，它占总耗时的 87% 到 89%。
- 字段计数。 在 GCC 16.2.0 上使用 `-std=c++20` 时，每项测试都会针对每个结构体调用库中的一个组件。仅字段计数（detail::fields_count）就再现了这种增长趋势：当 N = 500、1,000 和 2,000 时，其重载决议耗时分别为 1.47、4.42 和 15.83 秒；而 N = 2,000 时，整个 for_each_field 耗时 18.01 秒。已知字段数量后，将字段绑定起来（detail::tie_as_tuple）在 N = 2,000 时耗时 0.91 秒。

该库通过每种类型约十次聚合初始化探测来统计这 16 个字段（§2）。每次探测都是一个函数模板，其返回类型为 std::add_pointer_t<decltype(T{ubiq{I}...})>，并由替换失败并非错误机制保护，同时还重载了一个右值变体。每种类型只进行一次这样的探测，耗时就已呈二次增长：N 为 =500、1,000 和 2,000 时，分别耗时 0.30、1.05 和 4.21 秒；将其复制一份后，在 N 为 = 2,000 时耗时 4.15 秒。从副本中移除 std::add_pointer_t 包装后，耗时降至 0.97 秒；再移除右值重载和保护机制后，耗时降至 0.33 秒。这两种简化后的探测，从 N 为 = 1,000 增至 2,000 时，增长速度仍快于线性。我们未能确定 GCC 内部导致这一现象的原因，也没有测试 Clang。

随着字段数量增加，比率并非库的固定属性。在字段扫描（N = 50）中，Glaze 的传统后端 / P2996 后端比率在 M = 8时为 1.84，在 16 时为 1.85，在 32 时为 1.49，在 64 时为 1.05，在 128 时为 0.70（S2：0.70）；reflect-cpp 的比率在 16 时为 1.91，在 128 时为 2.51；Boost.PFR 的比率从 M = 32 起介于 2.86（M = 64）和 3.28（M= 128）之间，在 32 时为 2.90。从 M = 64 到 128，字段增加带来的成本增长幅度，Glaze 的 P2996 后端为 3.46×，而传统后端为 2.31×。在 M =128 时，P2996 后端的整个翻译单元（50 个结构体）工作集峰值达到 5.0 GB，而传统后端为 3.0 GB，因此内存开销随时间推移反而更高。一种与 Glaze 趋势相符的机制（我们未单独验证的假设）是：在 v9.1.0 中，其 P2996 后端会针对每个成员索引调用一次 nonstatic_data_members_of，既用于访问（member_info_at<T, I>），也用于获取名称（get_member_name_p2996<T, I>），因此其每个结构体的反射工作量会随着 M 增长得比线性更快。此后 Glaze 的主分支已更改这段代码（v9.1.0 之后的 PR #3112 [6], ），因此这一趋势是固定版本的特性。传统后端的开销也会随 M 超线性增长，因为它通过使用 1、2、...、M 个初始化项对聚合初始化进行探测，以确定成员数量（§2）。因此，这一假设探讨的是为何 P2996 后端的曲线弯曲得更明显。reflect-cpp 的 C++20路径在 M = 128时达到 8.4 GB（C++26 路径：3.1 GB）。在 M = 16 时，字段扫描的比率低于表 4 中的数值；表 4 中的比率是 N ≤ 200上的斜率比。后端比率也取决于 N：在 FrontendHeavy 中，Glaze 的边际成本比率在 N = 50 时为 1.86，在 N = 200时为 1.98；reflect-cpp 的相应比率为 1.81 和 2.05。

库与原生枚举的对比。 在同一个任务（FrontendHeavy，N ≤ 200中，原生分支每个结构体的成本为 1.39 毫秒，而表 3的成本为 1.72 毫秒，N ≤ 4,000 个结构体时，reflect-cpp 的 C++26路径成本为 10。55×，而 Glaze 的 P2996 后端为 9。17× 倍于原生分支的每结构体成本。两个库都会在我们调用的路径上计算字段名，尽管我们的序列化器会忽略这些名称——Glaze 对每个_字段_都会读取 reflect<T>::size，即 成员_名称<T>的大小，而 reflect-cpp 的 路径会构建_

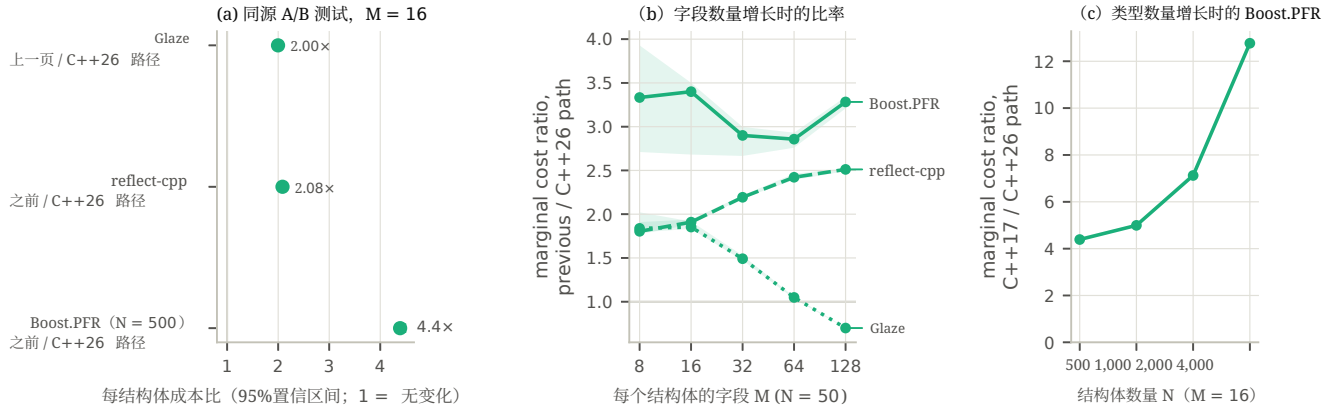


图 3。 先前路径 / C++26 每个库、标量语料库、会话 S1 的路径。(a) M = 16：对于 reflect-cpp 和 Glaze，这是 N = 0–200（FrontendHeavy，表 4）范围内每结构体斜率的比率；对于 Boost.PFR，这是 N = 500（FrontendN）处边际成本的比率；误差线表示 95% 自助法区间，竖线表示比率为 1。(b) N = 50（FrontendM）处的字段扫描：字段增加的成本，即 M 处的成本减去 M 处的成本 = 0，并带有 95% 区间；Boost.PFR 的 C++26 路径在 M = 8 和 16 处增加的计时器步长少于 10，因此其在这些位置的区间较宽。由于排除了与字段无关的成本，并且 N 固定为 50，M 处的数值 = 16 与 (a) 不同。(c) Boost.PFR 针对 N 的边际成本比率（FrontendN，M = 16）。

视图会构建一个命名元组——因此，这些比率衡量的是每个库为其字段访问器执行的每类型工作量，而不是基于库的反射本身的固有成本。对于每个结构体，Glaze 和 reflect-cpp 的 C++26 路径还大致将编译器峰值工作集的增长减半：Glaze 为 2,494 KB，而原生分支为 4,287 KB；reflect-cpp 为 2,846 KB，而原生分支为 6,228 KB。

完整翻译单元。按类型划分的比率不包括固定成本。在完整翻译单元中，切换到 C++26路径节省的时间较少，因为 Glaze 和 reflect-cpp 的头文件只需支付一次：在包含 50 个类型时，前端耗时的比例分别为 1。26× （Glaze）和 1。47× （reflect-cpp），而在包含 200 个类型时分别为 1。61× 和 1。84× （表 5）。

4.4 按类型划分的序列化器惯用法和重载集合（RQ3）

每种类型使用一个重载名称时，手写序列化器不再呈线性增长：从 N = 2,000 增至 4,000，手动方式的边际成本增加了 4.7 倍（不同名称：2.0 倍），在 N = 4,000 时达到 26.6 秒，其中编译器的重载决议计时器占 94%。在 N = 1,000 时，其边际成本是原生反射的 0.68× ，在 N = 2,000 时为 1.33×（相较于不进行注解查询的 P2996，分别为 0.92× 和 1.79×），因此在 N =1,000 到 2,000 之间的 span 形式中，它超过了原生反射。我们不再进一步缩小交叉区间：测得的增长超过二次增长，因此二次模型会错误地定位交叉点。使用一个重载名称的 X 宏表现类似（每次翻倍增加 ×4.7）。

在避免共享查找集合的惯用写法中，只有隐藏友元的增长比率区间不包含 2（2.2 [2.1，2.3]）；当 N 为 = 1,000 时，其开销是不同名称的 1.21× [1.16、1.27] 倍，远低于共享同一集合的惯用写法。

开销最高的惯用写法，是受enable_if约束的函数模板；这是 nlohmann::json v3.12 中NON_INTRUSIVE 宏的声明形式。它的开销远高于serialize模板；后者同样是函数模板，但没有约束（表 6）。当 N 为 = 1,000 时，重载决议计时器（它也会涵盖决议所触发的模板实参推导和替换）占其边际耗时的 93%，模板实例化占 26%，常量求值占 10%（普通自由函数分别为 82%、1% 和 0%）；其 GGC 内存增长了 815 MB，而普通自由函数为 135 MB。我们没有单独测定解析带约束候选项时哪一步开销最大。

调用点和候选项。 在固定的 K = 2，000 个重载集合中，假设每种类型都已在所扫描的调用之外被调用过一次（因此无论 C 为何值，序列化器主体都已实例化），使用自由函数时，每增加一个调用点的成本为 2.58 ms，使用序列化模板时为 3.56 ms，而使用独立名称时仅为 0.02 ms。在固定 C = 500 个调用点时，每增加一个重载的成本会随 K 增长：使用自由函数时，从 K = 500–1,000 时的 0.84 ms 增至 K = 2，000–4,000 时的 1.46 ms；使用序列化模板时，则从 0.96 ms 增至 1.67 ms；使用独立名称时则分别为 0.22 ms 和 0.23 ms。因此，该成本与调用点数量成正比，并且随同一查找集合中的候选项数量增长得快于线性增长；它并不是手写代码或类型数量本身的属性。一个在自身分派模板中执行非限定调用的库，会在模板实例化时为每种类型解析一次，因此它按类型而非按调用点付费：在每种类型都已实例化的情况下，每增加一个调用点的成本为 0.03 ms [0.01，0.06],与使用独立名称时一样低。

表 5. 前端的完整翻译单元（标量语料库，M = 16，FrontendHeavy，会话 S1）：编译器的总量及其峰值工作集，不含结构体，数量为 200。

Arm	总量，N = 0 (s)	总量，N = 200 (s)	峰值，N = 0 (MB)	峰值，N = 200(MB)
reflect_普通	0.31	0.59	93	140
pfr	0.28	0.56	86	163
rflcpp	0.90	7.10	252	1480
rflcpp_原生	0.83	3.86	209	783
glaze_经典	1.54	6.65	473	1301
glaze_原生	1.55	4.13	483	962

表 6. 十种定制惯用法，K = C = N（OverloadN）：N = 1,000 时的边际前端耗时、从 N = 500 增长到 1,000 时的增长情况（2 = 线性，4 = 二次），以及在 N = 1,000 时相对于不同名称的比率（对于共享同一查找集合的惯用法，该比率本身会随 N 增长）。

惯用法	记录形式由	在 N = 1,000 处每次	每翻倍 ×	N = 1,000 个独立名称
自由函数、限定调用	—	1.42	3.5	5.92
自由函数、参数依赖查找，直接调用	声明方式类似 nlohmann to_json，Boost.JSON 标签_调用，fmt format _as	1.41	3.5	5.88
库中的自由函数、参数依赖查找分派模板	nlohmann 的调用形式，Boost.JSON、fmt	1.48	3.5	6.17
由……保护的函数模板 enable_if（替换失败并非错误），参数依赖查找	nlohmann v3.12 非_INTRUSIVE	4.97	4.0	20.71
serialize（Archive&，T&，unsigned）模板	Boost.Serialization、cereal 外部	1.80	3.6	7.50
每种类型一个命名空间、参数依赖查找	—	0.25	2.1	1.04
隐藏友元（非模板）	BOOST_DESCRIBE_类; nlohmann v3.12 INTRUSIVE 声明一个由 enable_if 守卫的友元模板	0.29	2.2	1.21
成员函数	cereal 内部，protobuf	0.26	2.0	1.08
类模板特化	glz::meta、fmt::formatter	0.26	2.2	1.08
不同名称	—	0.24	2.0	1

-O2 下的 4.5 总计（RQ4）

在优化和代码生成阶段，若干总耗时并不随 N 线性增长（§3.6），因此我们在固定 N 下比较边际成本。在 N = 400 处，原生枚举的边际成本为 0.61×[0.60，Boost.PFR 的为（S2:0.61），P1061 的为 1.21×（S2: 1.22），而前端中的 P1061 为 7.96× 。在这些内联机制之间，后端会削弱前端差异，因为 GCC 的优化和代码生成阶段对每个内联分支耗时相同。在 N = 0 在 N = 400 时，手写代码耗时 2.16 s，X 宏耗时 2.19 s，P1061 耗时 2.19 s；原生枚举在不使用和使用注解查询时分别耗时 2.18 s 和 2.16 s。两个回调分支彼此相当：分别为 3.12 s 和 3.14 s。Boost.PFR 耗时 3.97 s，其 C++17 路径耗时 5.36 s。如果内联代码等价，那么优化器耗时相同正是预期结果；我们没有比较中间代码。与 Boost.PFR 相比，后端反而造成了差异。在优化器之外，两者耗时大致相同：在 N = 0上分别为 0.58 s 和 0.54 s。

在 N 为 = 400 时，优化器为 Boost.PFR 花费了 3.97 秒，而为原生分支花费了 2.18 秒。这一优化器耗时差距中，约有一半来自回调形式；无论使用 P2996 还是 P1061，它给优化器带来的成本都相同（分别为 3.14 秒和 3.12 秒）。在相同的回调 API 背后，原生枚举的成本为 0.85× [0.83，是 Boost.PFR 的 0.85]（S2：0.84）。仅回调形式一项，就使原生分支的边际成本增加了 1.38 倍。因此，原生内联分支在 -O2 下的大部分优势来自其 API 形式，而非枚举机制。其余部分则与 Boost.PFR 的逐字段分派器（§4.2）相符：每个字段索引都有一个特化供优化器内联。这是根据源代码得出的推断，因为我们没有检查中间代码。

这些比率会随 N 变化，因为优化器处理每个结构体的成本并非常数：在两个会话中，除 Boost.PFR 的 C++17路径外，FullN 中每个轻量臂在 N =100 处的单元格都高于穿过该臂其他单元格的直线。在 N = 50 、 100 、 200 和 400 时，原生 / Boost.PFR 分别为 0.68 、0.79、0.63 和 0.61——每个 N 下都低于 1；原生 / P1061 分别为 1.19、1.12、1.23

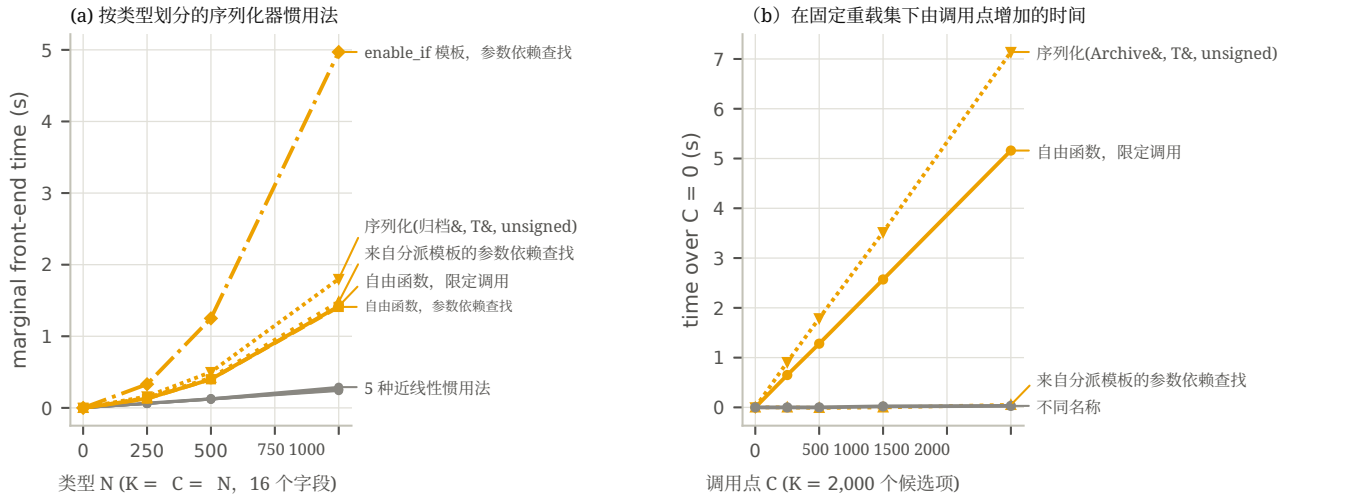


图4. 按类型划分的序列化器惯用法在标量语料库中的前端耗时，M = 16，会话 S1；对 11 轮测量取中位数，未绘制区间。(a) OverloadN，K = C = N：边际耗时，即 cost at N 减去 cost at N = 0，对应表6中的十种惯用法。边际耗时从 N = 500 增至 1,000 时至少变为三倍的五种惯用法以彩色绘制；限定自由函数调用、直接参数依赖查找调用以及分派模板中的参数依赖查找调用，在 N = 250处的差异均在一个 10 ms 计时器步长以内，在 N = 500 和 1,000 处则在 5% 以内，且彼此重叠。其余五种惯用法以灰色绘制。(b) OverloadC，K = 2,000 个重载，且每种类型都已调用一次：调用点增加的耗时，即 cost at C 减去 cost at C = 0。

和 1.21。Boost.PFR 的 C++17 路径成本是其 C++26 路径的 1.73× 倍（在 N = 50–400 范围内为 1.39–2.45× ）；注解查询原生分支的成本是具有不同名称的手写代码的 1.31× 倍，而使用一个重载名称时则是使用不同名称的 1.08× 倍。每个调用点使用一个函数时，注解查询分支相对于 Boost.PFR 的比率为 0.73×（使用一个函数时为 0.65× ），因此基准测试的布局降低了该比率，有利于原生分支，但并未改变其趋势。对于这些库，在 N = 25、50 和 100 时，Glaze 的传统 / P2996 比率分别为 1.27、1.36 和 1.48，而 reflect-cpp 分别为 1.25、1.50 和 1.64——在扫描末尾仍在上升；相比之下，前端处为 2.00 和 2.08。

4.6 更丰富的语料库、会话和编译器版本 (RQ4)

更丰富的语料库。对于字符串和容器（丰富语料库），原生枚举与 Boost.PFR 的成本再次大致相同：0.96× [0.95、1.03]。在此，每个库的 C++26 路径都比其先前路径成本更低：Glaze 低 2.06×，reflect-cpp 低 2.20× ，Boost.PFR 低 4.91×（其 C++17路径仍接近于 N ≤ 200 的线性增长，§4.3）。对于异构类型图（字段数量各异，包含枚举、嵌套、向量以及结构体可选值），原生枚举在 N ≤ 1=600 时的成本是 Boost.PFR 的 0.99×（HeteroN），在 N ≤ 200 时则是其 1.07×（HeteroHeavy）。与 P1061 相比，其成本为 1.55×（标量语料库：7.96×）。这一比率有所下降，因为该类型图为每个结构体增加的成本，与每个分支的成本相近。手写代码每个结构体耗时 1.98 ms，而标量语料库中为 0.23 ms；P1061 为 2.08 ms，而标量语料库中为 0.22 ms；原生枚举为 3.23 ms，而标量语料库中为 1.72 ms。绝对值原生枚举相较于手写代码的增量在不同语料库间变化不大：标量语料库中每个结构体为 1.49 ms，丰富语料库中为 1.21 ms，异构类型图中为 1.25 ms。然而，相对于手写代码的比率从 7.46× 降至 5.17× 和 1.64×。在该类型图上，各库的 C++26 路径仍然成本更低，但降幅不同：Glaze 后端的比率为 1.82，reflect-cpp 的仅为 1.21× [1.20、1.22]。Boost.PFR 的 C++17 路径在 N = =1,600 时，成本是其 C++26 路径的 4.37× 。在该类型图上，reflect-cpp 的 C++26 路径每个结构体耗时 77.09 ms，而标量语料库中为 14.66 ms（Glaze 的 P2996 后端：11.17 ms）。我们没有单独确定该类型图中的哪项特征导致 reflect-cpp 的成本上升。

GCC 16.1.0 与 16.2.0 对比（成对测试，轮次相同）。对于两个版本都使用 -freflection 编译的每个分支，GCC 16.2.0 都更快，但快的倍数各不相同，因此不同机制之间的比较取决于版本。在 CompilerN（N ≤ 2,000）中，原生分支上 GCC 16.1.0 每个结构体的开销是 GCC 16.2.0 的 1.124× 倍，P1061 上为 1.995× 倍，Boost.PFR 上为 1.550× 倍，手写代码上为 1.748× 倍。在这些库中（CompilerHeavy，N ≤ 200），差异较小：Glaze 的 P2996 后端上为 1.102× 倍，其传统后端上为 1.088× 倍，reflect-cpp 的 C++26 路径上为 1.065× 倍，C++20 路径上为 1.036× 倍。在 CompilerHeavy 中，库与原生分支的轮次相同；原生分支的比率为 1.247× [1.212，比每个库都高出 1.304]，而在 CompilerN 中为 1.124×（更高），因此原生分支的版本比率取决于扫描。在 CompilerN 中，GCC 16.1.0 下原生枚举每个结构体的开销分别是 Boost.PFR 的 0.71× 倍、P1061 的 4.16× 倍和手写代码的 4.44× 倍；GCC 16.2.0 下则分别为 0.99×、7.39× 和 6.91× 倍。注释

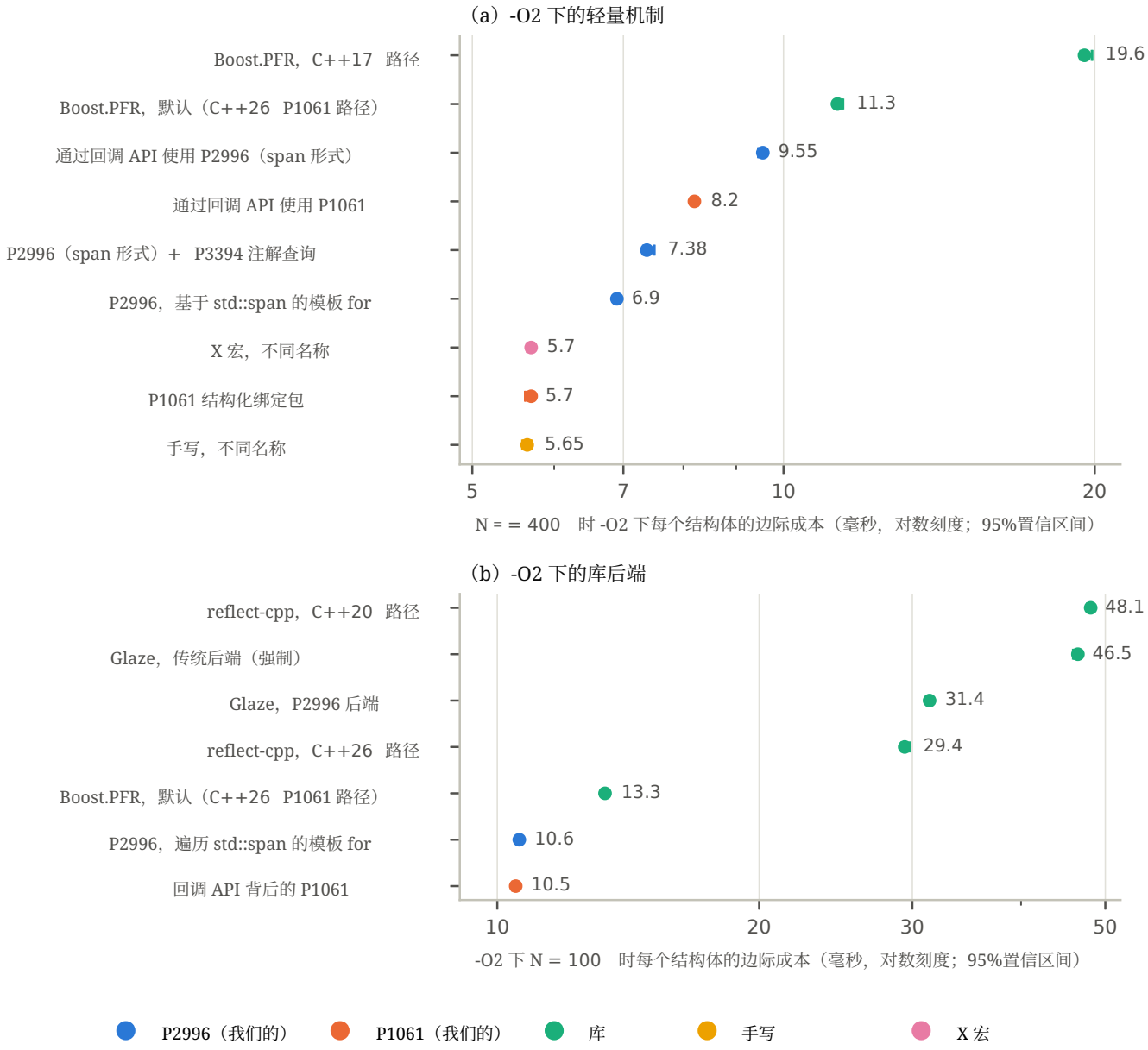


图5。每个结构体的成本（-O2，标量语料库，M = 16，会话 S1，对数刻度：最大 N 处的边际成本，即该处成本除以 N，而非斜率，因为有几条-O2 曲线并非线性（§3.6）；误差线表示 95% 自助法区间。P2996 分支使用清单 1 中的 span 形式。(a) N = 400 处的 FullN：轻型分支，包括 Boost.PFR 的 C 路径（手动和*_拆分控制组未列出）。(b) N = 100 处的 FullHeavy：reflect-cpp 和 Glaze 的两种路径，以及原生分支、Boost.PFR 和`p1061_visit` 作为参照。在 N = 100 处，轻型分支的每个结构体成本高于 N = 400 处（在 FullN 中，它们在 N = 100 处的数据点高于穿过其他数据点的直线，§4.5），而库的每个结构体成本仍随 N 增大而下降。

倍数（1.33× 和 1.33×）以及库后端 A/B 测试几乎没有变化：Glaze 的比率在 GCC 16.1.0 上为 1.93，在 16.2.0 上为 1.95；reflect-cpp 分别为 1.97 和 2.03。对于 Boost.PFR 和 P1061，版本差异仅在使用 -freflection 时才会出现。不使用该选项编译时，两个版本的每个结构体开销相同（GCC 16.1.0 / 16.2.0：1.000 和 0.990）。在相同轮次中，该标志

Boost.PFR 的耗时为 1.703，P1061 的耗时为 2.578（使用 GCC 16.1.0）；而使用 GCC 16.2.0 时，耗时分别为 1.092 和 1.264。GCC 16.1.0 未启用该标志时，手写代码无法编译。我们未能确定 GCC 16.2.0 中是哪项更改导致了这一差异。GCC 的 C++ 前端变更日志列出了两个针对不同版本间仅供 consteval 使用的类型的检查的更改：PR125179 和 PR

。125280 [51]。

会话。 S2 在不同日期使用相同的编译器和代码运行了相同的计划。对于标题数字背后的逐类型斜率，S2/S1 比率介于 0.972 （ GCC 16.1.0 上的 P1061，CompilerFlag ）和 1.020（Boost.PFR，FlagCost）之间；在 S2 中，P1061 在 GCC 16.1.0 上的标志开销为 2.48× ，而 S1 中为 2.578× 。在该计划中所有通过线性检查且至少增加十个计时器步长的 102 个逐分支斜率中（§3.5 的 96 个，以及 ReportOverhead 的六个外部墙钟时间斜率），会话之间差异的中位数为 0.7%，90% 的差异至多为 2.5%，最大为 8.8%（字段扫描中原生分支的逐字段斜率，1.088 [1.066，1.140]; 没有任何标题数字使用该斜率）。会话内区间并未覆盖会话之间的差异：在这 102 个 S2/S1 比率中，有 11 个的区间排除了 1；如果区间覆盖该差异，预计约有 5 个会如此。仅凭两个会话无法判断这是日期之间的变异，还是来自 11 轮测试的百分位区间覆盖不足；无论如何，几个百分点的分支间差异都应结合两个会话来解读，而不应仅对照一个区间。非斜率的标题数字同样具有良好的复现性：在 S2 中，Boost.PFR 在 N = 500 时的 C++17 / C++26 比率为 4.5，在 N = 4,000 时为 12.8，而 Glaze 在 M = 128 时的比率为 0.70。原生枚举与 Boost.PFR 在 M = 128 时的比率无法复现（S1 中为 0.80，S2 中为 0.89），因此我们不使用该比率。合并两个会话后，主要比率为 7.88（原生 / P1061）、1.03（原生 / Boost.PFR）、1.37（注解查询）、1.99（Glaze）和 2.08（reflect-cpp）。

4.7 可维护性（RQ5）

每类型源代码是分析性计数，并非测量结果。对于 16 个字段，手写序列化器每种类型需要 19 行代码，X 宏字段列表需要 18 行（每行一个字段）：两者都会为每种类型将每个字段命名一次，因此都会按 O(N·M) 增长，而 P2996、P1061 和库分支则不需要针对每种类型编写源代码。一次性通用代码方面，原生序列化器需要 28 行（加上注解查询及其标签类型则为 42 行），P1061 需要 21 行，每个库适配器也需要 21 行（不包括库本身）；X 宏共享代码为 25 行。

5 相关工作

仅使用原生反射。 Romeo 在 GCC 16 上使用原生 P2996 的 [42] 次测试（预发布版、hyperfine；以 -O0 编译并链接单个翻译单元）：在发布版构建中，包含 <meta> 的开销约为 155 毫秒，每种反射类型则为 1.1–1.7 毫秒。他最初的数据来自启用了断言的编译器；根据不同项目，重新进行发布版构建后速度快了约 20–50%。他的后续测试 [41] 在 GCC 16.1.1 上比较了反射、枚举库和用于枚举到字符串转换的 X 宏，其中 <meta> 的开销占主导。两者都没有比较库或手写字段枚举，而且计算每个翻译单元的总开销时都包含了固定的 <meta> 开销；我们的每种类型数据则将其排除在外，因为构建时，我们测得 <meta> （§4.1）的开销为 30 毫秒；这是在翻译单元已包含 <string>、 <vector> 和 <optional> 的基础上增加的开销，而不是相对于一个空翻译单元而言。

并行测量。 字段反射库 [31]（自 2026-10-12 起在 README 中提供表格）在 GCC 16 上使用 -std=c++26-freflection 时切换到 std::meta 后端，并报告了访问带名称的 8 字段结构体所增加的编译时间（同一翻译单元中 10 到 200 种类型之间的斜率，-c -O0，3 次取最佳）：在 GCC 16 上，其 C++26后端为 20 毫秒，而 Boost.PFR 的 for_each_field_with_name 为 18 毫秒；在 GCC 15 上，其 C++20 后端为 23 毫秒，而 Boost.PFR 为 22 毫秒。它的两个后端是在不同编译器上计时的，且未提供区间，也未与 P1061、手写或宏基线进行比较。mp-units [39] 报告称，在一个工作负载上，其基于反射的转换引擎耗时约为 −10% 至 −12%，在一个 ISQ 占比较高的翻译单元上，其值域表达式运算耗时为 +8% 至 +33%（在精简工作负载上则几乎没有差异）；测量仅在 WSL 上进行，作者提醒：“正负方向可信，具体幅度则不可信。” reflect-moc 对 moc 替代品 [44] 的测量发现，在计算机负载较高时，总耗时比 moc 低约 12% 至高约 6%。simdjson 报告称，启用反射会导致性能下降 6%（CppCon 2025 [28], clang-p2996）；之后，它通过更改每个 TU 实例化的内容，将反射示例的构建时间缩短了 35% [26]。Kolpackov（GCC PR125329 [25]）测量了在使用和不使用头文件单元时，跨 128 个 TU 的 <meta> 开销。Matek [30] 发现，在 Clang 分支上，基于反射的类型标识比基于模板的类型标识快约 4× 。GCC 自身的报告则跟踪了启用该标志的开销，以及反射密集型常量求值的开销：PR124925 [33]（GCC 16.0 中对 <bits/stdc++.h> 的开销高出 19%，之后以已修复结案）、PR 125179 [29] （GCC 16.2.0 中性能下降 40 倍，后降至 1.1×），以及 Sajdak（PR128900–PR128902 [43, 45, 46],，2026年11月），其统计了 16.2.0 前端执行的指令数：identifier_每个字符约产生 3.5 万条指令，因为返回的字符串_视图是通过 char_traits 循环测量的。reflect-cpp 的维护者借助 Clang 的 -ftime-trace [48], 发现，主导其构建开销的是序列化代码生成，而不是反射；我们的库分支测量的是字段访问器，而非序列化器，因此无法检验这一点。

库。 Glaze 将其 P2996 后端描述为一项新能力；我们未找到关于该后端的公开编译时 A/B 对比数据。我们找到的唯一一项 Glaze 编译时数据（make_keys_info<20>，v7.7.0 发行说明 [4] 中的 45.1 → 26.0 毫秒）与 P2996 后端无关，且使用 -ftime-trace 测得；在 v9.1.0 之后，Glaze 的主分支更改了其 P2996 后端枚举成员的方式（PR #3112 [6]）。reflect-cpp 的文档声称，其 C++26 路径在没有数据 [49], 的情况下编译成本更低，而其持续集成不会构建该路径。Boost.PFR 附带一个实验性的 P2996 引擎（自 Boost 1.91.0 起），但其 for_each_field 在 -std=c++26 下不会使用该引擎。Glaze PR #2848 [7] （已于 2026-08-29 合并）出于许可原因替换了其元组实现，并且

随之引入了一个隐藏在 get 后面的 N 候选项 operator[] 重载集；在元组微基准测试中，它报告编译时间降低了 −41%，而在整个 glaze.hpp 翻译单元中降低了 −1% 至 −3%（Apple clang21）。这是我们找到的、最接近 §4.4 重载集效应的实践证据；我们针对六个序列化和格式化库的定制惯用法进行测量，并分别改变调用点和候选项的数量。

已采用，未测量。 使用 P2996 进行序列化或代码生成的演讲和库报告的是功能能力，而非编译成本：Webber [52], Piper [35], Chen [10],simdjson 5.0 [27].

测量编译时间。 Metabench [15] 通过减去基线编译时间来隔离代码块，正如我们计算边际成本的做法；Chochlík（P2560R0 [14]）代表 SG7 比较了基于值和基于类型的反射 API 的编译成本：在生成的源代码中模拟多达一百万个元对象，并在当时尚无任何 P2996 实现之前减去基线编译时间；ctbench [34] 使用 Clang 的 -ftime-trace 测量扩展趋势，并指出比较元编程策略“没有明确的方法论”；作为模板实例化调试器的 Templight [38], 后来被用于分析模板实例化 [37]。Gregor 和 Järvi [18] 在 GCC 上将一种新的语言特征——可变参数模板——与其预处理器模拟进行比较，并按阶段拆分结果，这是与本研究最接近的先例。我们的方案遵循基准测试文献中的做法：随机交错试验 [1],、关注设置偏差 [32], 、报告带置信区间的效应量 [23], 、通过重采样核验轮次数（§3.6）、计算自助法区间 [2, 17], , 并记录各项因素 [19]。

6 效度威胁

构念效度。 一种原生形式。S3 将成员查询（§4.2）与另外三种表述进行了比较，但只在前端、GCC 16.2.0 上、针对含 16 个字段的标量语料库，并且只在一次会话中进行。丰富型和异构型语料库、字段扫描、-O2 和 GCC 16.1.0 的所有原生数值，以及注解和回调分支（reflect、reflect_visit）的所有数值，都采用 span 形式，因此包含了遍历 span 的成本；S3 的前端对比表明，在 GCC 16.2.0 上，如果使用常量数组，这些前端数值会更低，但我们并未对此进行测试。

检测工具与标志。*_noreport 与 *_noflag 对照用于测量其他分支所使用的两个选项的成本，分别为 -ftime-report 和 -freflection（§3.1、§4.1）。若不使用该报告，原生分支相对于手写代码的比率在外部墙钟时间上大约会高出 9%，而其相对于 Boost.PFR 的比率则不会改变（§4.1）；我们并未测量该报告对 P1061、X 宏或其他分支的开销，因此它对相对于 P1061 的主要比率的影响尚不明确。由于只有阶段计时器对一次编译进行划分，而在其余计时器中只有两个栈计时器相互排除彼此（§4.2），因此我们逐一报告计时器占比，并且只将两个栈计时器相加。该标志被 GCC 标记为实验性的，也会拖慢不使用反射的分支。与未使用该标志编译的那些分支相比——如同未启用反射的项目——原生分支的每结构体比率大约会按各分支对应的该标志系数增大（§4.1）。

内部效度与结论效度。 混杂因素。 代码相同的对照实验表明，翻译单元解析的内容可能使每类型成本变化几个百分点（§4.2），因此，原生实现方案之间若存在这种量级的差异，就不能归因于具体机制；这些方案会解析<meta>，而 P1061、手写和 Boost.PFR 方案则不会。我们没有测试 <meta> 本身是否会产生这种影响，也没有测试 pfr_native 的命令行中多出的一个 -D 是否造成设置偏差（§5）。如果 <meta> 确实有此影响，那么相对于这些方案的原生比率就会被高估几个百分点：这足以影响 span 形式与 Boost.PFR 的相当性，但不会影响更大的比率。将 span 与数组之间的差距主要归因于迭代展开，而非定义_static_array 时增加的未计时调用，是一种推断（§4.2）；贡献 2 和 §7 都将其表述为推断。

在主要分析完成后新增的会话。S1 和 S2 的结果经过分析和审查后，才新增了 S3（§3.5）。其分析规则预先规定了哪些对比可以用于归因，而不规定结果。这些规则在会话开始前设定，但当时七个分支中已有六个（除后来添加的数组形式外）都曾以非正式方式各编译过一次；加入数组形式后，也对整个任务进行了一轮冒烟测试。两者都未用于分析，但冒烟测试已显现出 span/数组差异。共享分支复现了 S1 的结果，而 span/数组因素造成的两倍差异远大于 S1 和 S2 之间的波动（§4.2）；但由于 S3 只有一个会话，其区间低估了日间波动，而且所有数组形式的结果都仅基于 S3。

计算机与会话。一个任务的各分支共用轮次，并在漂移哨兵的编译之间按随机顺序运行；它们均在高性能核心上以高优先级运行（§3.4、§3.5），因此负载和漂移只会增加噪声，而从期望上看不会造成分支之间的偏差。剔除负载超过 15% 时进行的编译后，斜率变化很小；在 S1 和 S2 中，长时间编译几乎没有脱离 CPU 运行（S3 的情况稍多；§3.5）。不过，编译器内部的墙钟时间对负载很敏感；测试所用计算机是一台普通台式机，常用应用均处于开启状态，且 Microsoft Defender 的实时保护也已开启（§3.7）。此外，无法排除频率变化：电源计划固定为“高性能”并有记录（§3.5），但未记录核心频率。会话内区间未涵盖 S1 和 S2 之间的全部变异（§4.6）。

外部效度。 单一编译器系列。关键分支也在 GCC 16.1.0 上进行了计时（§4.6），Boost.PFR 的 C++17path 则在 GCC 15.2.0 上进行了探测，其二次项也同样显现（§4.3）；我们没有测试 Clang。截至 2026年11月，GCC 16 是唯一已发布且支持 P2996、能够

生成目标代码的编译器：Clang 上游仅支持基本类型 [9]；彭博的 clang-p2996 [8] 是一个自称实验性的分支；EDG 7.0 [16]（自 2026年9月起开放源代码）会通过生成 C 代码的后端运行 P2996 代码，因此其计时结果会包含 C 编译；而在 Compiler Explorer 上，它拒绝了一个使用展开语句（模板 for，P1306）的探测程序，我们的原生实现方案都使用了该语句。因此，所有数值都是 GCC 16 实现的特性，并且可能随次版本变化：从 GCC 16.1.0 到 16.2.0，机制比率的变化远大于库后端 A/B 测试（§4.6）。不使用 `-freflection`时，P1061 和 Boost.PFR 在 GCC 16.1.0 与 16.2.0 上的开销相同，因此两个版本间的差异出在该标志上；这与 GCC 16.2.0 的 `-freflection` 修复（包括 PR125179 和 PR125280）相符。我们没有对此进行二分定位；对于需要该标志的原生分支，无法区分具体原因。

标准与实现。 标准规定，span 形式的展开语句逐个元素进行迭代，而数组形式则会解构数组（数组类型的表达式永远不能用于展开迭代）；两者各自的开销取决于 GCC 16.2.0 和 libstdc++ 的 define_static_array，这一特性不一定适用于其他编译器、库或版本。GCC 16.2.0 还将 span 形式的人工变量声明为静态变量，而 CWG3044 [20] 已从规范中移除了这一说明；因此，span 形式会受到块作用域静态变量的二次注册开销影响（§4.2，PR128928 [22]）。一项于 2026-11-16 [21], 提交至 gcc-patches 的补丁截至 2026-11-21 尚未合并，该补丁会为 C++26移除这些静态变量。我们预计该补丁会消除 span 形式的这部分开销，但尚未对此进行测量。

语料库与布局。这些语料库涵盖了多种类型形状（§3.2），但既不能复现真实代码库，也无法界定其范围：没有语料库包含基类、用户定义类模板或私有成员，字段名也只有两到四个字符。原生分支不会查询任何名称，但 Glaze 和 reflect-cpp 的 C++26 路径会对每个字段调用 identifier_of；在 GCC 16.2.0 上，该调用的开销会随着名称长度增加（PR{v5）。它们的前置 C++26路径则会从 __PRETTY_FUNCTION__ 中解析名称。更长的名称可能会改变库分支的开销，甚至可能影响其后端比率，但不会影响原生实现方案。在 -O2 下，通过一个函数序列化每个结构体，对注解分支有利，相较于 Boost.PFR 更占优势，但不足以改变总体趋势（§4.5）；第 7 节中 -O2 比率所涉及的分支（reflect_plain、reflect_visit、p1061）无法控制布局。

构建方式、主机与硬件。两款编译器的构建方式相同（§3.7）：未经过引导构建，启用了发布版检查，由 MSYS2 主机上的 GCC 15.2.0 构建，并采用 configure 的默认标志（GCC 16.2.0 的构建日志中为 -g -O2）。三阶段引导（GCC 的默认方式），或发行版可能添加的配置文件反馈或链接时优化，都可能让编译器运行得更快，但各阶段的提速幅度未必相同，因此绝对耗时可能有所不同；耗时落在不同阶段的分支之间，其比率也可能随之变化。前端代码在所有主机上都相同，但在这台 Windows 主机（MinGW-w64）上，configure 未检测到匿名内存映射（已针对 GCC 16.2.0 的构建进行检查），因此 GCC 的垃圾回收器从 malloc 页面组获取内存页，而不是通过 mmap 获取。与采用不同的微架构或内存系统一样，这也可能改变工作集相差数倍的分支之间的比率（§4.2，§4.3）。我们没有在 Linux 或其他计算机上使用经过引导构建的编译器重新进行测量。

范围。 库分支计时的是各个库的字段访问器（§3.1），而非序列化器；后者对每种类型所做的工作更多（键、格式化）。我们的原生分支和 P1061 分支不会计算字段名（二进制格式不需要字段名；P1061 无法提供字段名），而 Glaze 和 reflect-cpp 的访问器会计算（§4.3），因此就这一点而言，原生分支与 Glaze 和 reflect-cpp 的比率并非同类比较，且有利于原生分支（Boost.PFR 的 for_each_field 不计算字段名）；像 JSON 这样的基于名称的格式会要求原生序列化器添加 identifier_of，代价则未纳入我们的测量。我们既未实现也未测量反序列化。另有两种形式未计时：每个索引都重新计算的原生成员列表，如 Glaze v9.1.0 中的 P2996 后端（§4.3）；以及无需反射、直接解构结构体本身的展开语句（§2）。

7 结论

在 GCC 16.2.0 上，对含 16 个标量字段的结构体进行原生 P2996 枚举，在会话 S3 中计时的四种写法下，前端成本为每个结构体 0.82 至 2.02 毫秒。P2996R13 自身的多个示例所采用的形式，是对 define_static_array 返回的 std::span 使用展开语句。对 $N \le 4{,}000$ 个结构体使用这种形式时，成本为每个结构体 1.72 毫秒（S3：1.73），是 P1061 结构化绑定包的 7.96× 倍；而 P1061 结构化绑定包的成本本身与手写序列化器相当，后者为每种类型各自定义名称。span 形式约一半的成本来自 span；我们认为这主要是因为 GCC 需要计算其上的迭代展开语句，而 C++26 对此有规定（§4.2）。对常量数组进行相同的循环，成本要低 0.47×。内联编写时，span 形式的成本约为 Boost.PFR 默认情况下每_个_字段的成本；使用相同的回调 API 时，成本则为其 1.21× 倍。对于常量数组，内联原生枚举的成本为 Boost.PFR 每_个_字段成本的 0.49× 倍；我们没有对回调形式的数组方案进行计时。对每个成员进行 P3394 注解查询，会使 span 形式的成本增至 1.37 倍。在我们的三个语料库中，与手写代码相比，span 形式使每个结构体的前端工作增加 1.21–1.49 毫秒。因此，其比率取决于结构体其他方面的成本：在异构类型图上，它是 P1061 的 1.55× 倍；在 -O2 下则为 1.21× 倍，此时优化器在每种内联机制上花费的时间大致相同。在 -O2 下，内联原生序列化器的成本为 Boost.PFR 的 0.61× 倍，主要是因为回调 API 会增加优化器的工作量。使用相同的回调 API 时，该比率为 0.85。

在 GCC 16.2.0 上的实践建议。在整个翻译单元范围内，库的选择比枚举机制的选择影响更大。对于 200 个含 16 个标量字段的结构体，使用内联原生枚举时，前端耗时 0.59 秒；使用 Boost.PFR 时为 0.56 秒。reflect-cpp 和 Glaze 的 C++26 路径还会由访问器计算字段名，耗时分别为 3.86 秒和 4.13 秒（表 5）。对于内联原生序列化器，在前端使用对常量数组的展开语句 `[:std::meta::reflect_constant_array(...):]` (standardC++26) 是更省成本的选择（每个含 16 个标量字段的结构体，成本为 span 形式的 0.47× 倍，S3）；计入固定成本后，整个翻译单元在含 1,000 个结构体时耗时为 span 形式前端时间的 0.61× 倍，含 4,000 个时为 0.50× 倍。我们没有在其他语料库或 -O2 下对数组形式计时；在-O2 下，含 400 个结构体时，span 形式的成本已经只有 P1061 的 1.21× 倍（§4.5），因此数组形式节省的前端开销在这种情况下可能影响较小。对于不需要名称或注解的代码，例如处理二进制格式、哈希或比较，可以使用 P1061 包枚举结构化绑定能够拆解的结构体字段。对我们的序列化器而言，这种做法的成本与手写代码相当，而且无需为每种类型编写源代码。我们没有对另一种标准形式计时，即用于解构结构体的展开语句。若某个库允许用户通过在一个命名空间中重载同一名称来自定义序列化，那么每次解析该名称时都必须考虑该命名空间中的所有重载，因此成本会随着类型数量的增加而以快于线性的速度增长（§4.4）。隐藏友元、成员函数和类模板特化则可避免共享的重载集合（表 6）。所有每类型成本数据均不包括固定成本；预编译头文件或模块或许可以摊销这些成本，但我们都没有进行测量。每类型成本会在序列化该类型的每个翻译单元中重复产生。

对于每种类型，在 16 个字段时，每个库的 C++26 路径都比其 pre-C++26 路径更容易编译。Glaze 的提升为 2。00×，reflect-cpp 的提升为 2。08×，Boost.PFR 的提升为 4。4× 在 500 种类型时为 12。8×在 4,000 种类型时也是如此，因为其 C++17 路径会随着类型数量增加而近乎二次方增长。但这种提升并非库的固定属性。字段数为 128 时，Glaze 的结果反转（0。70×，固定版本 v9.1.0）；reflect-cpp 的降至 1。21×（在异构类型图上）。在整个翻译单元中，头文件会削弱这种提升。对于表 6 中共享同一查找集合的五种惯用写法，使用同一名称的手写序列化器，其类型数量从 500 增至 1,000 时，每翻倍一次，增长 3.5–4。0×（K = C = N）。使用单一重载名称的手写方案增长更快：类型数量从 2,000 增至 4,000 时（§4.4），并且在 span 形式下，类型数量介于 1,000 和 2,000 之间时，其成本超过原生反射。

工具链也会改变比较结果。在 GCC 16.2.0 及更高版本中，-freflection 标志会使不使用反射的代码每种类型的开销增加 1.017–1.264×；在 GCC 16.1.0 中，开销则最多增加 2.578×。而在这两个次版本之间，span 形式的内联原生枚举开销从 0.71× 变为 0.99×，与 Boost.PFR 的开销相当；库后端 A/B 测试的变化则微乎其微。因此，不同机制之间的比较取决于编译器版本和所测量的具体实现方式；随着 GCC 的反射支持日趋成熟，以及其他编译器陆续提供支持，应当重新进行比较。

制品

该制品已归档于 https://doi.org/10.5281/zenodo.24443567（本文所报告的版本），开发地址为
h
tt
ps://github.co
m/DenysKachalenko/CPP26FieldEnumerationCompileCostArtifact；其代码采用 MIT 许可证发布，数据和文本采用 CC BY 4.0 发布。 `DATA.md` 记录了原始数据中的每个文件和每一列。所有数值和图表均可根据归档原始数据重新生成，无需编译器，只需使用 Python、NumPy 和 Matplotlib（此处版本分别为 3.13.1、2.5.1 和 3.11.0；`requirements.txt`），从Harness/运行：

- python papernumbers.py ../Data/Sessions/ S1 ../Data/Sessions/S2 ../Data/Sessions/S3 会写入 Data/PaperNumbers.json；
- python paperlint.py 会将 Paper.md 中的每个标注数字与其进行核对；
- python figures.py ../Data/Sessions/S1 ../Data/Sessions/S3 会绘制 Figures/ 中的图表。

重新测量需要两种编译器，它们由 Harness/ build_gcc.sh 构建；每个会话还需进行一次无人值守运行（Harness/overnight.sh、RunBook.md）。S1 和 S2 的编译各耗时约 5.7 小时，S3 的编译约耗时 16 分钟。归档包含生成器（Harness/generate.py、Harness/overload.py）、臂注册表和操纵检查（Harness/arms. py、Harness/preflight.py）、Harness（Harness/ campaign.py、Harness/compiler.py、Harness/machine. py、Harness/plan.py）、无人值守运行脚本（Harness/ overnight.sh）、Boost.PFR 探针（Harness/pfrprobe. py、Data/PfrProbe.json）、统计数据和论文数字（Harness/stats.py、Harness/papernumbers.py、Harness/paperlint.py、Harness/figures.py、Data/PaperNumbers.json、Figures/）、GCC 构建脚本（Harness/build_gcc.sh）、固定版本信息（Deps.json）、CMake 测试、PR128928 的复现用例（Repro/）、每次运行的原始数据、环境记录以及每个会话所测代码的副本（Data/Sessions/S1、S2、S3），还有运行手册（RunBook.md）。

致谢

作者构思了这项研究，确定了研究设计和方法学决策，进行了测量并核查了结果。在作者的指导下，Anthropic 的 AI 助手 Claude 被用于编写测量测试框架和分析脚本、起草和编辑文本、检索文献以及审阅论文。测试框架本身并不被视为可信：值预言机通过阴性对照，将每个实验组的输出与手写参考实现进行比对；§3.3 中的操纵检查则核查每一个单因素

组合，完整方案则在两次会话中进行了测量。论文中的每一个测量数值均由分析脚本根据归档原始数据生成，并自动与文本核对（制品）。作者对内容承担全部责任。

参考文献

[1] Ali Abedi 和 Tim Brecht。2017。于高度多变的云计算环境中开展可重复实验。载于第八届 ACM/SPEC 国际性能工程会议论文集（意大利拉奎拉）（ICPE '17）。计算机协会，美国纽约州纽约市，287–292。doi:10.1145/3030207.3030229[2] Edd Barrett、Carl Friedrich Bolz-Tereick、Rebecca Killick、Sarah Mount 和 Laurence Tratt。2017。虚拟机预热时冷时热。《ACM 编程语言学报》1，OOPSLA，文章 52 2017 27 10 1145 3133876（10 月），页码。[3] 2026 *++26* *2996* doi: 。/ Stephen Berry。。C P 反射支持。检索于 2026年11月20日，网址：https://github.com/stephenberry/glaze/blob/v9.1.0/docs/p2996-reflection.md Glaze文档，docs/p2996-reflection.md，v9.1.0。[4] Stephen Berry。2026年。Glaze v7.7.0。检索于 2026年11月20日，网址：https: //github.com/stephenberry/glaze/releases/tag/v7.7.0 发行说明，发布于 2026-05-21。[5] Stephen Berry。2026年。Glaze v9.1.0。检索于 2026年11月20日，网址：https: // github.com/stephenberry/glaze/releases/tag/v9.1.0 发行 v9.1.0，提交 43b175e。[6] Stephen Berry。2026年。在 P2996 反射中包含基类成员。检索于 2026年11月20日，网址：https://github.com/stephenberry/glaze/pull/3112 Glaze 拉取请求 #3112，于 2026-11-18 合并。[7] Stephen Berry。2026年。全新的 glz::tuple 实现，不再沿袭 tuplet。检索于 2026年11月20日，网址：https://github.com/stephenberry/ glaze/pull/2848 Glaze 拉取请求 #2848，于 2026-08-29 合并。[8] 彭博。2026年。clang-p2996：支持 WG21 P2996（反射）的实验性 Clang 实现。检索于 2026年11月20日，网址：https://github.com/ bloomberg/clang-p2996 GitHub代码库（llvm-project 的分支），开发分支 p2996。[9] changkhothuychung。2026年。 *[clang]*：支持基本类型的 std::meta::info。检索于 2026-11-22，网址：https://github.com/llvm/llvm- project/pull/190356 LLVM 拉取请求 #190356，于 2026-10-02 合并。[10] Jolly Chen。2026年。C*++26* 及后续版本的编译期元编程。doi:10.17181/30bhp-wad72 口头报告幻灯片，“高能物理领域中 C++26 及后续版本的编译期元编程”，第 28 届高能与核物理计算会议（CHEP 2026），2026 年 5 月 26 日。[11] Wyatt Childers、Peter Dimov、Dan Katz、Barry Revzin、Andrew Sutton、Faisal Vali 和 Daveed Vandevoorde。2025年。C*++26*反射。技术报告 P2996R13。ISO/IEC JTC1/SC22/WG21。https://wg21. link/p 2996 13 r [12] Wyatt Childers、Peter Dimov、Barry Revzin 和 Daveed Vandevoorde。2025年。define_static_{string,object,array}。技术报告 P3491R3。ISO/IEC JTC 1 22 21 21 3491 3 /SC / WG 。https://wg .link/p r [13] Wyatt Childers、Dan Katz、Barry Revzin 和 Daveed Vandevoorde。2025年。反射注解。技术报告 P3394R4。ISO/IEC JTC 1 22 21 21 3394 4 /SC /WG 。https://wg .link/p r [14] Matúš Chochlík。2022。。比较基于值的反射与基于类型的反射。技术报告 P2560R0。ISO/IEC JTC1/SC22/WG21。https://wg21. link/p 2560 0 r [15] Louis Dionne 和 Bruno Dutra。2019。Metabench：用于编译期微基准测试的简易框架。检索于 2026年11月20日，网址：https: // github.com/ldionne/metabench 软件，GitHub代码库，master 分支，提交 3322ce7（2019-09-16）；无带标签的发行版。[16] Edison Design Group。2026年。EDG C++ 前端，7.0 版。检索于 2026-11-22，网址：https://github.com/edgcpp/compiler 开源发布版本，github.com/edgcpp/compiler 的 7.0 标签（提交 51d5795e），2026年9月25日。[17] Bradley Efron 和 Robert J. Tibshirani。1993年。《自助法导论》。《统计学与应用概率专论》第57卷。Chapman & Hall，纽约。[18] Douglas Gregor 和 Jaakko Järvi。2008年。C++0x 的可变参数模板。《对象技术期刊》第8卷第88期第2期（2008年2月），31–51。doi:10.5381/jot. 2008.7.2.a2 特刊：SAC 2007 的 OOPS 专题。[19] Torsten Hoefler 和 Roberto Belli。2015年。并行计算系统的科学基准测试：报告性能结果时告知大众的十二种方法。载于《高性能计算、网络、存储与分析国际会议论文集》（得克萨斯州奥斯汀）（SC '15）。计算机协会，美国纽约州纽约市，文章 73, 12 10 1145 2807591 2807644 页。[20] 2025 doi: . / . Jakub Jelinek。。迭代展开语句的问题。ISO/IEC J TC1/SC22/WG21 核心工作组。检索于 2026-11-22，网址：https://cplusplus.github.io/CWG/issues/3044.html CWG 议题 3044，提交于 2025-07-08，在 2025年11月会议上获接受（状态 C++26）。[21] Jakub Jelinek。2026年。 *[补丁]* c++：修改 C*++26* 的迭代展开语句，使其不再使用 static for。gcc-patches 邮件列表。检索于 2026- 11-22，网址：https://inbox.sourceware.org/gcc-patches/avrTHDVsBbB-7fhD@tucnak/ 发布于 2026年11月16日。[22] Denys Kachalenko。2026年。大量块作用域静态变量导致编译时间呈二次增长：确定_局部_判别符扫描翻译单元范围列表。GCC Bugzilla。检索于 2026-11-21，网址：https://gcc.gnu.org/bugzilla/ show_bug.cgi?id=128928 错误 128928（组件 c++），提交于 2026-11- 21。[23] Tomas Kalibera 和 Richard Jones。2013年。《在合理时间内进行严格基准测试》。载于《2013年内存管理国际研讨会论文集》（美国华盛顿州西雅图）（ISMM '13）。计算机协会，美国纽约州纽约市，63–74。doi:10.1145/2464157.2464160[24] Dan Katz、Andrew Sutton、Sam Goodrick、Daveed Vandevoorde 和 Barry Revzin。2025年。展开语句。技术报告 P1306R5。ISO/IEC JTC 1 22 21 21 1306 5 /SC /WG 。[25] 2026 https://wg .link/p r Boris Kolpackov。 . 使用 import std 时 C++ 反射的编译速度比使用头文件包含更慢。GCC Bugzilla。检索于 2026-11-20，网址：https://gcc.gnu.org/bugzilla/show_bug.cgi? id=125329 错误 125329（组件 c++），提交于 2026-05-15。[26] Daniel Lemire。2026年。不要在每个翻译单元中实例化碎片化的_json。检索于 2026-11-20，网址：https://github.com/simdjson/ simdjson/pull/2894 simdjson/simdjson 中的 GitHub 拉取请求 #2894，于 2026年9月30日合并。[27] Daniel Lemire。2026年。simdjson 5.0 发布。检索于 2026-11-20，网址：https://lemire.me/blog/2026/09/28/simdjson-5-0-is-out/ 博客文章，2026年9月28日。[28] Daniel Lemire 和 Francisco Geiman Thiesen。2025年。用于 JSON 序列化的 C*++26* 反射：实践之旅。检索于 2026-11-20，网址：https://simdjson.github.io/simdjson_ _ 2025_ 2025 _ ++ talks/cppcon2025/cppcon slides.html 演讲幻灯片，CppCon 。演讲视频：“C 终于如何在 JSON 序列化方面胜过 Rust”，https://www.youtube.com/watch?v=Mcgk3CxHYMs。[29] Edoardo Lolletti。2026年。启用 -freflection 标志后构建速度极慢。GCC Bugzilla。检索于 2026-11-20，网址：https://gcc.gnu.org/ bugzilla/show_bug.cgi?id=125179 错误 125179（组件 c++），提交于 2026-05-04，已在 GCC 16.2 中修复。[30] Ivan Matek。2025年。回复：C*++26* 反射与 Boost。检索于 2026- 11-20，网址：https://lists.boost.org/archives/list/boost@lists.boost.org/message/Z5ZULHHJBBRYM2QJ6AKJSDTPPGOHSMJS/ 发至 Boost 开发者邮件列表的帖子，2025年8月6日。[31] Yoshiki Matsuda。2026年。field-reflection：用于 C++ 的编译时反射，可从结构体/类中获取字段名和类型。检索于 2026-11- 20，网址：https://github.com/yosh-matsuda/field-reflection 软件，

版本 v1.1.1（2026年10月28日）；README.md 中的编译时表，首次添加于提交 0d0d6fe（2026年10月12日）。[32] Todd Mytkowicz、Amer Diwan、Matthias Hauswirth 和 Peter F. Sweeney。2009年。什么都没做却产生错误数据，而且看起来完全没错！载于《第14届编程语言与操作系统体系结构支持国际会议论文集》（美国华盛顿特区）（ASPLOS '09）。计算机协会，美国纽约州纽约市，

265 276 10 1145 1508244 1508275 – 。

[33] 2026 doi: . / . Patrick Palka。 . -freflection 会使编译速度变慢（例如 stdc++.h），幅度约为 20%。GCC Bugzilla。检索于 2026-11-20，网址：https://gcc.gnu.org/bugzilla/show_bug.cgi?id=124925 错误 124925（组件 c++），提交于 2026年4月18日，于 2026年9月14日解决，状态为 FIXED。[34] Jules Penuchot 和 Joel Falcou。2023年。ctbench——编译时基准测试与分析。《开源软件杂志》第8卷第88期（2023年8月），。doi: . /joss. Callum Piper。 . 使用 C++ 反射生成语言绑定。检索于 2026-11-20，网址：https://github.com/CppCon/CppCon2026/blob/main/Presentations/Generating_Language_Bindings_using_Cpp_Reflection.pdf 演讲幻灯片，CppCon 2026，2026年9月17日。[36] Antony Polukhin。2026年。Boost.PFR（Boost 1.92.0）。检索于 2026-11-20，网址：https://github.com/boostorg/pfr/tree/boost-1.92.0 7.0 标签 boost-1.92.0，提交 401385c。[37] Zoltán Porkoláb、József Mihalicza、Norbert Pataki 和 Ádám Sipos。2009年。C++ 模板元程序性能分析技术研究。《布达佩斯大学科学年鉴》，计算机科学分部 30（2009），97–115。https://ac.inf.elte.hu/Vol_030_2009/doi/097_30.pdf [38] Zoltán Porkoláb、József Mihalicza 和 Ádám Sipos。2006年。调试 C++ 模板元程序。载于《第5届生成式编程与组件工程国际会议论文集》（美国俄勒冈州波特兰）（GPCE '06）。计算机协会，美国纽约州纽约市，

255 264 10 1145 1173706 1173746 – 。doi: .

[39] 2026 / . Mateusz Pusz。 . 更新 C++26 反射 A/B 测试套件，并以原生方式确认其两项发现。检索于 2026-11-20，网址：https://github.com/mpusz/mp-units-benchmarks/issues/10 mpusz/mp-units-benchmarks 中的 GitHub问题 #10，2026年9月14日开启。[40] Barry Revzin 和 Jonathan Wakely。2024年。结构化绑定可以引入一个包。技术报告 P1061R10。ISO/IEC JTC1/SC22/WG21。https://wg

21 1061 10 .link/p

[41] 2026 r Vittorio Romeo。 . 枚举到字符串的成本：C++26 反射与传统方法。检索于 2026-11-20，网址：https://vittorioromeo.com/index/ blog/refl_enum_to_string.html 博客文章，2026年5月12日。[42] Vittorio Romeo。2026年。C++26 反射隐藏的编译时成本。检索于 2026-11-20，网址：https://vittorioromeo.com/index/blog/refl_ compiletime.html 博客文章，2026年3月6日；之后以发布构建的 GCC 数据修订（勘误）。[43] Filip Sajdak。2026年。常量求值中的 std::vector<T>(n) 对平凡类型T的每个元素都通过函数调用进行构造和销毁：成本是 new T[n]() 的 4.5 倍。GCC Bugzilla。检索于 2026-11-20，网址：https://gcc. gnu.org/bugzilla/show_bug.cgi?id=128902 错误 128902（组件 libstdc++），提交于 2026-11-20。[44] Filip Sajdak。2026年。Qt层的测量结果。Wro.cpp。检索于 2026-11-20，网址：https://github.com/wrocpp/reflect-moc/blob/857c72f762045c3030f8513eac4f719bdbbdcb04/docs/measurements.md reflect-moc 仓库（wrocpp/reflect-moc）中的 docs/measurements.md，提交 857c72f，2026年10月28日。[45] Filip Sajdak。2026年。 [反射] 标识符的常量求值工作量为 O(length)：尽管 reflect.cc 知道字符串长度，却仍从指针构造 string_view。GCC Bugzilla。检索于 2026-11-20，网址：https://gcc.gnu.org/bugzilla/show_bug.cgi?id=128900 错误 128900（组件 c++），提交于 2026-11-20。[46] Filip Sajdak。2026年。字符串内置函数在 constexpr堆 或 自动存储上不会进行常量折叠，因此常量求值中的 char_traits 循环占反射密集型代码前端工作量的 22%。GCC Bugzilla。检索于 2026年11月20日，来源：https://gcc.gnu.org/bugzilla/show_bug.cgi?id=128901 错误 128901（组件 c++），提交于 2026年11月20日，判定为错误 94082 的重复项。[47] Tristan Soliven。2026年。《C++26 反射：它对大型代码库究竟有何改变》。Whole Tomato Software。检索日期：2026年11月20日，来源：https://www.wholetomato.com/blog/cpp26-reflection-large-codebases/ 博客文章，2026年5月1日。[48] Patrick Urbanke。2025年。加速反射。检索日期：2026年11月20日，来源：https://github.com/munich-data-engineering/reflect-cpp/pull/537 reflect-cpp 拉取请求 #537，维护者于 2025年11月3日发表的评论。[49] Patrick Urbanke。2026年。C++26 反射。检索日期：2026年11月20日，来源：

h

t

t

p

s://

git

hub.com/munich-data-engineering/reflect-cpp/blob/v0.27.1/docs/content/cpp26_reflection.md reflect-cpp 文档，v0.27.1 版本中的 docs/content/cpp26_reflection.md。[50] Patrick Urbanke 等。2026年。reflect-cpp v0.27.1。检索日期：2026年11月20日，来源：https://github.com/munich-data-engineering/reflect-cpp/releases/tag/v0.27.1 v0.27.1 版本发布，提交 e0ee3f6。[51] Lewis Walton。2026年。 [反射] -freflection 标志会导致类模板参数中的类成员在引用其所属类型时出现不完整类型错误。GCC Bugzilla。检索日期：2026年11月20日，来源：https://gcc.gnu.org/bugzilla/show_bug.cgi?id=125280 错误 125280（组件 c++），提交于 2026年5月12日，2026年5月13日修复。[52] Andy Webber。2026年。《C++26 反射能为您做什么？一次侧重于二进制协议支持的反射应用实践实验》。检索日期：2026年11月20日，来源：https://github.com/CppCon/CppCon2026/blob/main/Presentations/What_Can_Cpp26_Reflection_Do_For_You.pdf 演讲幻灯片，CppCon 2026。