The Complete Guide to
Speed Up Your C++ Builds

C++构建加速指南

INCREDIBUILD

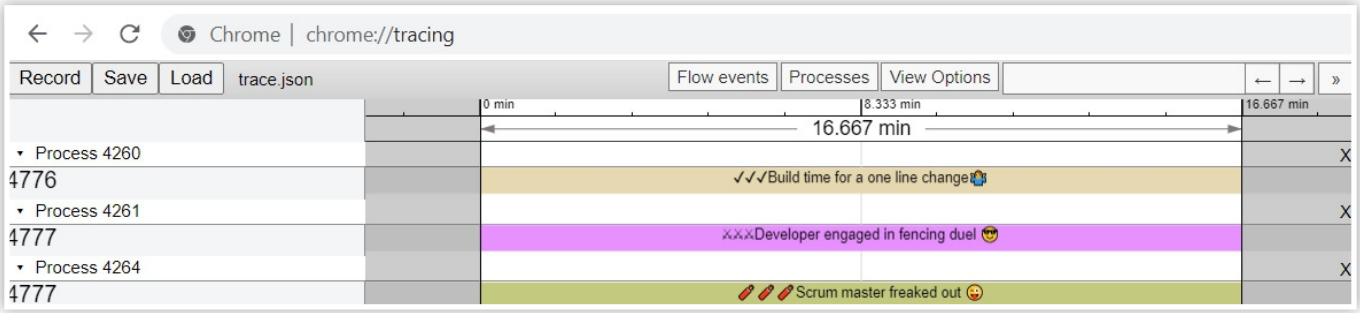
目录

C++ 构建的疑难杂症

C++ 是一门伟大的语言，我们都是忠实粉丝。但 C++ 的构建时间确实是个难题。如果你正在使用 C++ 进行构建，那么你的构建大概率都耗时冗长，这对你、你的经理和整个团队来说，都是不小的挑战。

这个问题并不新鲜，可以说很常见。尤其是在敏捷工作环境中，持续集成/持续部署（CI/CD）已成规范。你肯定不会想为一个简单的签入耗费一整个下午的时间，对吧？

很多人选择无为而治。还有一些人利用这些漫长的构建时间进行休闲娱乐（像 XKCD 击剑笑话描述的那样）。然而，还有些人认为这个问题值得关注。大家想想我们，我们是哪一类人呢？



（以上图片为假设情况，如与你当前项目有任何雷同之处，纯属巧合）

为什么 C++ 构建耗时长？

很多 C++ 开发人员都在思考这个问题。以下列出了几个原因，以及相关的解释：

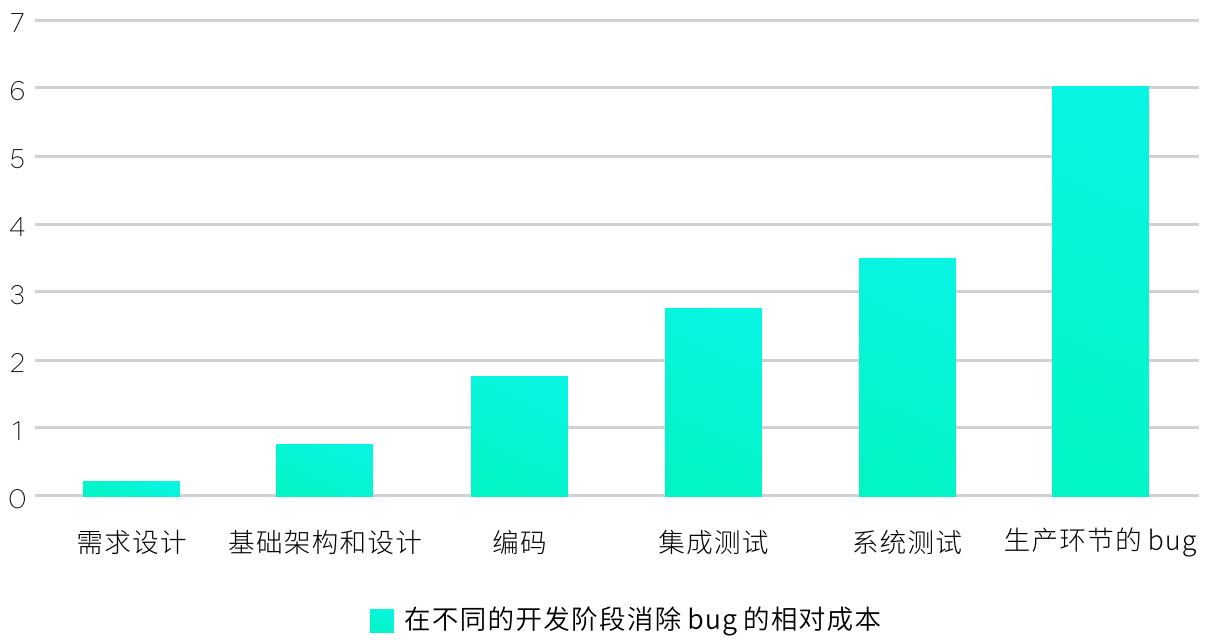
- 构建机器资源不足
- 构建依赖项太多
- 构建使用过时的编译器/链接器
- 构建未使用预编译头文件，或使用错误
- 要求编译器进行最佳优化
- 代码库未维护/镀金代码
- 代码库庞大

大量需要解析的头文件是原因之一，但如上所述，这并不是唯一的原因。一般来说，编译过程中处理数据的复杂动态方式是构建时间冗长的主要原因。

为什么耗时构建是个大问题?

很显然,这是一个棘手的问题。在最新的 C++ 调查中,超 40% 的开发者报告,构建时间是一个主要问题,大约 40% 以上的开发者认为这是一个小问题。只有 17% 的人根本认为这不算问题。等待构建完成,或根据时间优先安排开发进程(每周构建),同时延迟测试并跳过任务以避免编译,这可能会对组织产生破坏性影响。

受影响的不仅仅是交付时间。忽视大型构建的时间问题,甚至会影响软件的根本质量。



这个图表流传甚广,列举了在不同阶段检测出 bug 并修复的成本。如果因构建时间太长而牺牲测试,那么自然会有越来越多的 bug 逃到生产环节。这将增加开发的总体成本,软件的质量也会相应受损。在当前竞争激烈的市场中,高质量的频繁发布至关重要。

左移的势头正盛。如果开发者在将代码提交到存储库之前,就可以运行测试,那么软件的外部质量将得到提高。如果同时进行静态代码分析,软件的内部质量也会夯实。

开发者的效率也会因构建时间冗长而受损。如果构建反馈需要更长的时间,那么开发者改进代码的思绪也会阻滞不前。这些都是构建时间过长带来的直接和间接成本。

缩短 C++ 编译时间真正有效的方法

在本指南中，我们将介绍各种缩短 C++ 编译时间的方法。

我们为什么要这样做？我们不是在加速编译吗？编写一份指南，介绍这个问题的其他解决方案，不会影响我们的自身业务吗？

我们坚信，解决构建时间难题可以多管齐下。

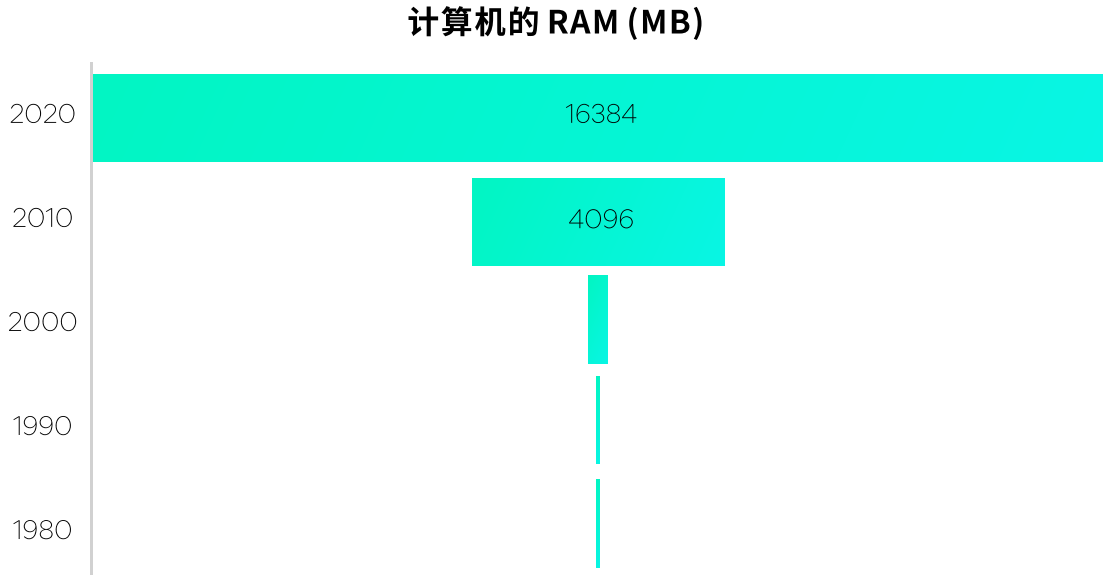
事实上，本指南中介绍的解决方案都是有效的，在某些情况下可以发挥真正的价值，尽管在发挥作用之前可能需要一些准备工作。当然，使用 Incredibuild 加速构建是独辟蹊径的正解。

话不多说，我们直入主题。

升级构建机器

毫无疑问，投资优质硬件是一个简单直接的解决方案。如果不考虑购买更好的硬件，根本无法讨论减少 C++ 构建时间的问题。更多的 RAM、更好的硬盘和更好的 CPU 都可以缩短构建时间。

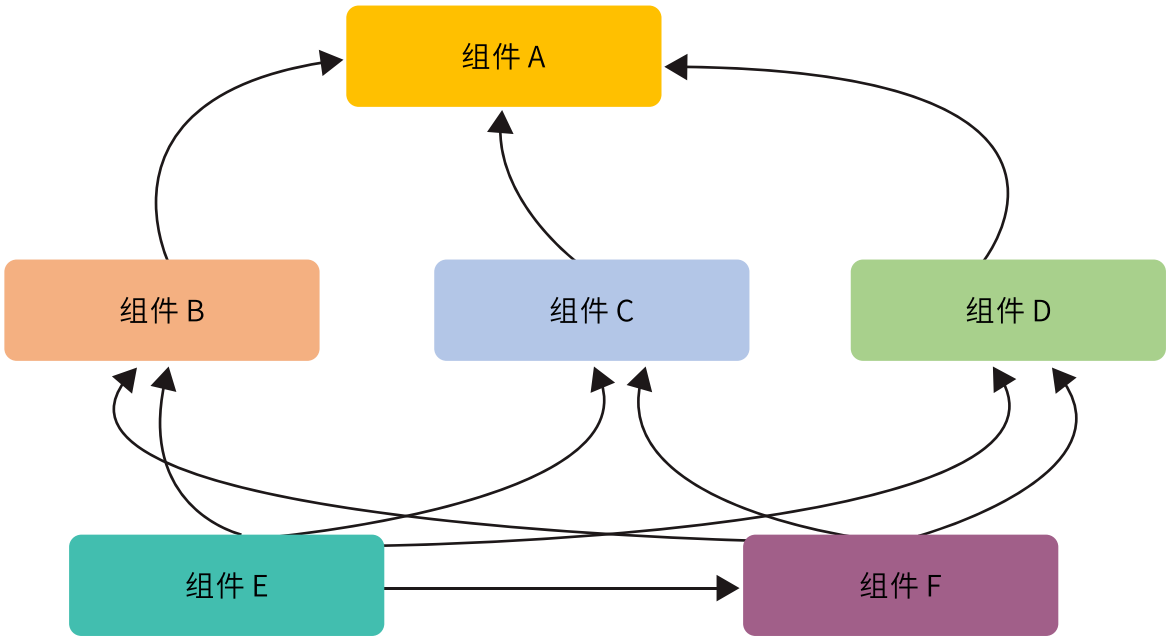
计算机 RAM 的摩尔定律：



减少依赖项

文件、组件、模块和层之间的紧密耦合拉长了构建进程。如果项目的设计图沿着这些线（方框里也可以是文件、模块或层），那么问题就出现了。

问题是：是否需要所有这些依赖项？其中哪些可以轻松解除？哪些解耦的工作量或风险最大？请一位架构师来分析成本和收益，并对项目进行重新设计。随着内部代码质量的提高，编译时间将减少。这是一种通用技术，不仅限于 C++ 项目。



紧密耦合的系统让人望而生畏

让我们用一个运行示例来展示我们在本指南中讨论的一些问题，展示如何缩短构建时间。
首先，我们引入一些头文件：

Lyrics.h

```
#pragma once
namespace SongLibrary
{
	class Lyrics
	{
		// The code for Lyrics is complicated...
	};
}
```

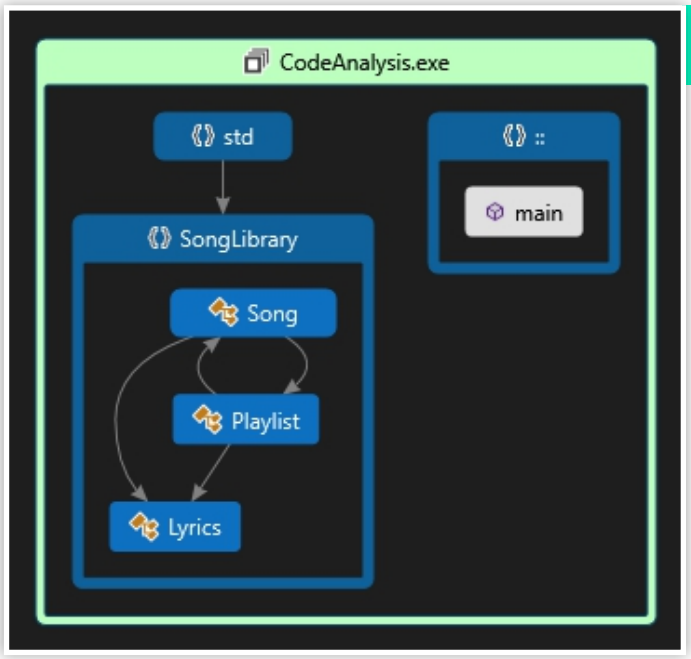
Song.h

```
#pragma once
#include <memory>
#include <string>
#include <vector>

namespace SongLibrary
{
	class Lyrics;
	class Playlist;
	class Song
	{
	private:
		std::wstring m_Writer;
		std::wstring m_CoAuthor;
		std::shared_ptr<Lyrics> m_Lyrics;
		int m_YearOfRelease;
	public:
		int GetYearOfRelease() const { return m_YearOfRelease; }
		std::wstring GetWriter() const { return m_Writer; }
		std::wstring GetCoAuthor() const { return m_CoAuthor; }
		// Constructor for clients still using classic C++
		Song(std::wstring writer, std::wstring coauthor, const
Lyrics□ const lyrics, int yearofrelease);
		// Constructor for clients of modern C++
		Song(std::wstring writer, std::wstring coaauthor,
std::shared_ptr<Lyrics >lyrics ,int yearofrelease;)
		// Design Decision. Copy and Move allowed. No assignment
		Song(const Song&) = default;
		Song& operator(=const Song = )&delete;
		Song(Song = )&&default;
		// Design Creep. Unfortunately we have a requirement
		std::vector<std::weak_ptr<Playlist>> m_Playlists;
	};
}
```

Playlist.h

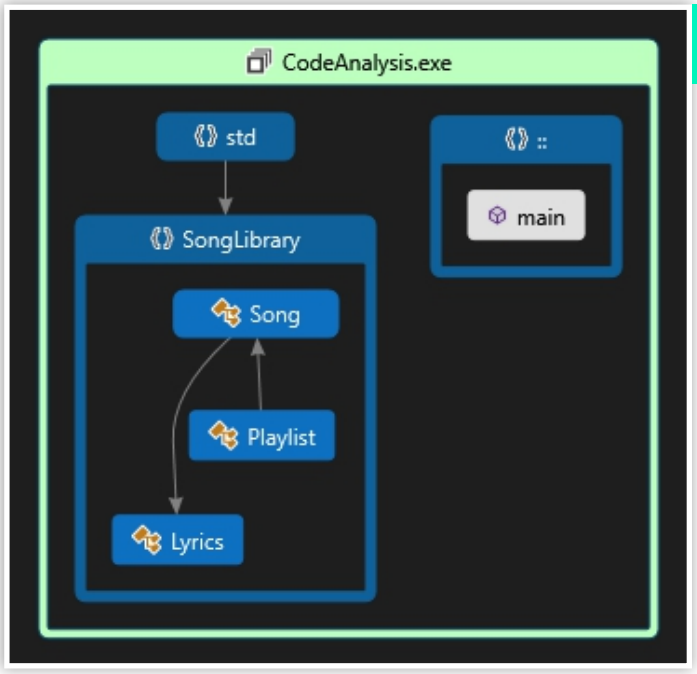
```
#pragma once
#include "Song.h"
#include "Lyrics.h"
#include <vector>
#include <chrono>

namespace SongLibrary
{
      class Playlist
      {
      private:
  s          td::vector<Song> m_Songs;
            std::chrono::duration<int> m_TotalPlayingTime;
  s          td::shared_ptr<Lyrics> m_LyricsOfCurrentSong;
      public:
  s          td::wstring getPlayingTime();
            bool RemoveSongFromPlaylist(Song toberemoved);
            bool AddSongToPlayList(Song tobeadded);
      };
}
```

这个系统设计看起来怎么样?



Not a good design

你看到上图中 Song 和 Playlist 之间的循环依赖关系了吗？
系统实际应该是什么样子？



合理的设计

当你改进系统设计时，将从本质上改善构建时间。
（请了解我们的构建监视器和可视化工具，以便更轻松地分析瓶颈和依赖项）。此外，当你改进设计时，我们将在下一节中看到，改善构建时间的其他方案也会变得更加简便直接。

静态 VS 动态链接

这个方法可能针对特定平台，但所有现代操作系统都有一种动态链接到代码的方法。

- 动态链接库（DLL Windows）
- 动态加载模块（dylib Macintosh）
- 动态加载的库（DL Linux）

更改较少的实用程序可以调入动态库。由于这些代码不再跟随系统的每个构建进行重复编译，因此可以大大缩短编译时间。

再次进行这样的更改需要重构和重新设计。不可避免的成本，是此类动态代码的版本控制。当 DLL 的接口发生变化时，最好分配一个新版本。较为流行的版本控制方案如下所示：

主要版本	次要版本	构建号	修订号

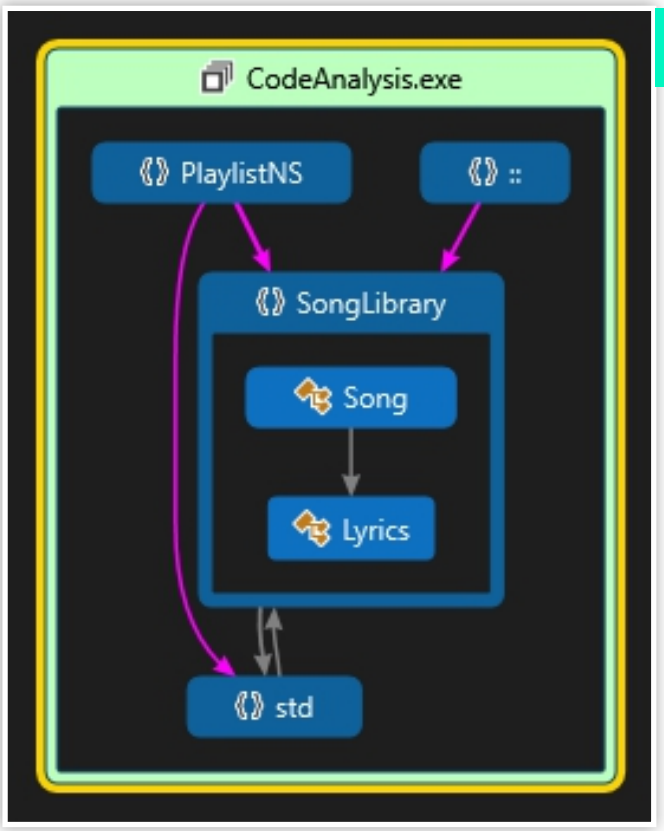
如果客户使用不同版本的产品，则 DLL 的主要版本号将是不同的。跨产品版本维护会产生成本，但从 DLL 构建的角度来看，比静态链接要好。

我们在上一节中介绍了一个运行示例，直观地看到了缩短时间的方法。现在让我们再进一步改进设计。首先，我们注意到 Playlist 必须从 SongLibrary 名称空间分离到一个独立的名称空间。我们这样操作：

Playlist.h (已更改)

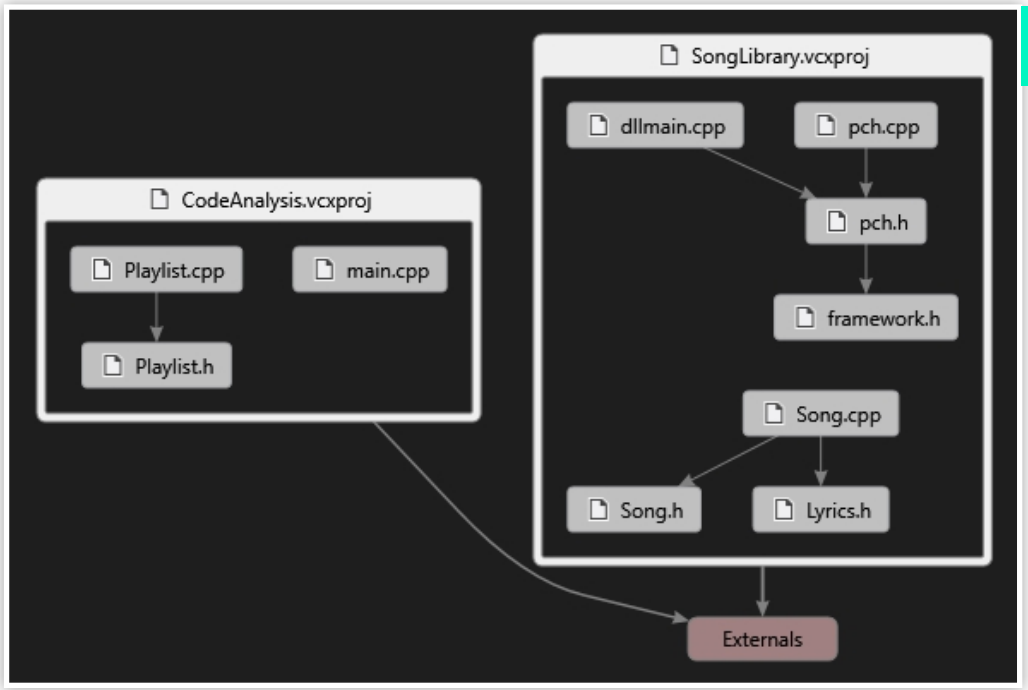
```
#pragma once
#include "Song.h"
#include "Lyrics.h"
#include <vector>
#include <chrono>

namespace PlaylistNS
{
      class Playlist
      {
      private:
  s          td::vector<SongLibrary::Song> m_Songs;
            std::chrono::duration<int> m_TotalPlayingTime;
            std::shared_ptr<SongLibrary::Lyrics>
m_LyricsOfCurrentSong;
      public:
  s          td::wstring getPlayingTime();
            bool RemoveSongFromPlaylist(SongLibrary::Song
toberemoved);
            bool AddSongToPlayList(SongLibrary::Song tobeadded);
            Playlist();
      };
}
```

我们看到了什么？



组件更加清晰

很明显，我们可以将 SongLibrary 中的所有内容分离成一个独立的库。我们看到 SongLibrary 没有太大变化。所以我们将其分离到一个动态库中。



DLL 模块化

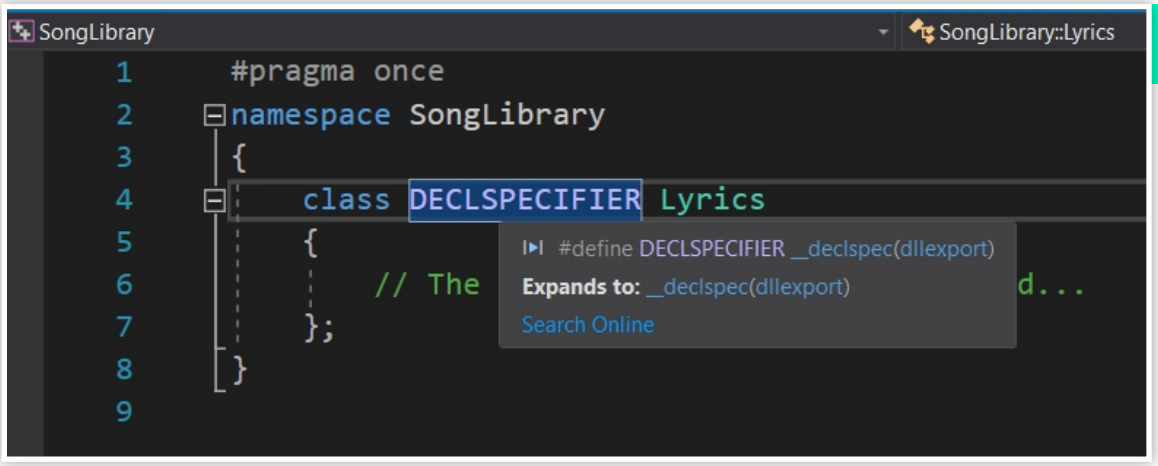
我们将在下一节中讨论新的 pch.h/pch.cpp 文件。让我们专注于框架。文件。以下是它的特点：

Framework.h

```
#pragma once

#ifdef SONGLIBRARY_EXPORTS
#define DECLSPECIFIER __declspec(dllexport)
#define EXPIMP_TEMPLATE
#else
#define DECLSPECIFIER __declspec(dllimport)
#define EXPIMP_TEMPLATE extern
#endif
```

这是 Windows 导出 DLL函数的机制。
让我们看看其他文件是如何因创建单独的 DLL 而发生变化的。

Lyrics.h
(已更改)



可以看到，Lyrics 类是从 DLL 导出的。对 Song.h 也必须进行类似的更改。但我们也注意到出现了维护警告：

Severity	Code	Description	Project	File	Line
Warning	C4251	SongLibrary::Song::m_Writer': class 'std::basic_string<wchar_t,std::char_traits<wchar_t>,std::allocator<wchar_t>>' needs to have dll-interface to be used by clients of class 'SongLibrary::Song'	CodeAnalysis	D:\Demo\Song Library\Song.h	13

```
#include <vector>
#include "framework.h"

namespace SongLibrary
{
      class Lyrics;

EXPIMP_TEMPLATE template class DECLSPECIFIER
std::shared_ptr<Lyrics>;

      class DECLSPECIFIER Song
      {
      private:
        .          ..
```

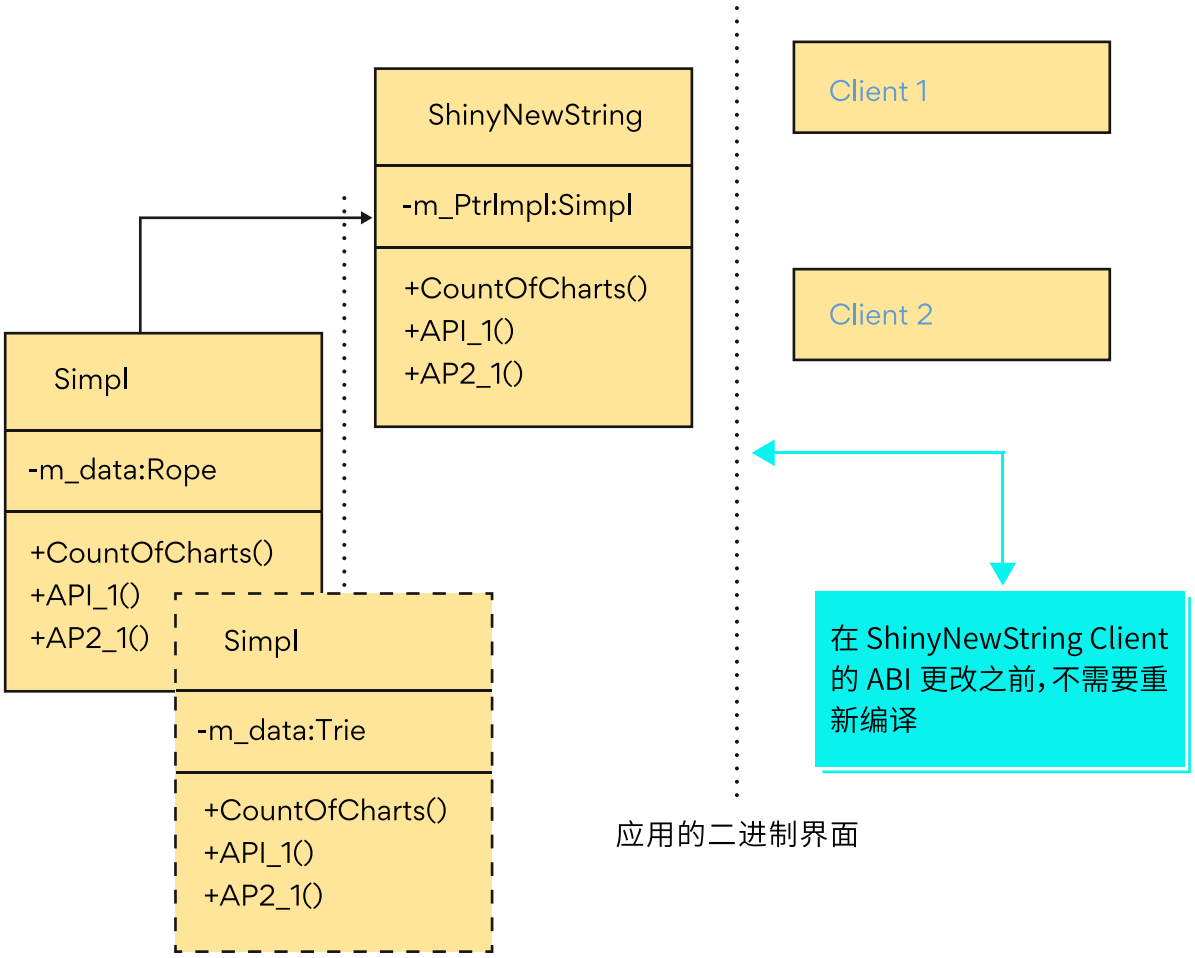
出现了维护警告

可以看到，我们被迫生成 std::shared_ptr<Lyrics> 类的所有文件。为什么？因为 DLL 接口上的 STL 类不是一个好的设计选择。我们能怎么办？输入 Pimpl！

静态 VS 动态链接

PImpl 是一种备受欢迎的技术，通过减少类之间的依赖项来缩短 C++ 项目的构建时间。它也被称为编译时防火墙，因为它阻止编译器查看实现的细节。代码实现可能会改变，但由于使用该类的客户端接口保持不变，因此不必重新编译。这大大提高了性能。

让我们以需要设计 ShinyNewString 类为例



由于设计使用指针进行代码实现，所以内部实现的更改对类的外部客户端是不透明的。这正是该技术也称为编译时防火墙的原因。

毫无例外，这个方法也需要权衡利弊，在使用 pImpl 时，牺牲的是性能。当类的成员函数被委托给底层实现类时，必须有一个间接级别来执行类的成员函数。代码变得有点复杂，代码的可测试性也降低了但是 pImpl 是一种很好的技术，可以减少 C++ 项目的构建时间。

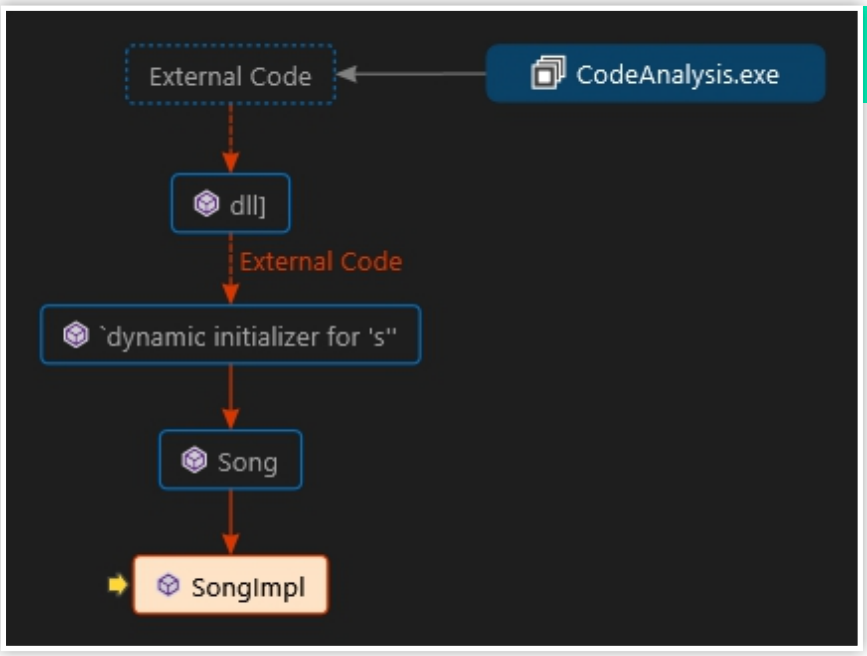
让我们回到运行示例，用代码说明如何使用 pImpl idiom 来优化设计。

我们有这个交互界面：

Song.h (使用 STL)

```
class DECLSPECIFIER Song
{
      private:
            std::wstring m_Writer;
            std::wstring m_CoAuthor;
            std::shared_ptr<Lyrics> m_Lyrics;
            int m_YearOfRelease;
public:
...
```

我们将其更改为：



对 Song 类使用 PImpl 设计

代码看起来是这样:

Song.h (使用 pImpl)

```
#pragma once
#include <memory>
#include <string>
#include <vector>
#include "framework.h"

namespace SongLibrary
{
      class Lyrics;
      class SongImpl;
      class DECLSPECIFIER Song
      {
      private:
            SongImpl* m_songImpl;
      public:
            int GetYearOfRelease() const;
            std::wstring GetWriter() const;
            std::wstring GetCoAuthor() const;
            // Constructor for clients still using classic C++
  S          ong(std::wstring writer, std::wstring coauthor, const
Lyrics* const lyrics, int yearofrelease);
            // Constructor for clients of modern C++
            Song(std::wstring writer, std::wstring coauthor,
std::shared_ptr<Lyrics> lyrics, int yearofrelease);
            // Design Decision. Copy and move allowed. No assignment.
  S          ong(const Song&); // Can no longer be default. Why?
            Song& operator=(const Song&) = delete;
            Song(Song&&); // Can no longer be default. Why?
  ~          Song(); // Naked pointer member needs a destructor.
      };
}
```

仅用于说明
请在生产代码中使用智能指针。

最后，我们得到 SongImpl 类:

SongImpl.h

```
#pragma once
#include <string>
#include <memory>

namespace SongLibrary
{
	class Lyrics;
	class SongImpl
	{
	private:
		std::wstring m_Writer;
		std::wstring m_CoAuthor;
		std::shared_ptr<Lyrics> m_Lyrics;
		int m_YearOfRelease;
	public:
		int GetYearOfRelease() const { return  m_YearOfRelease; }
		std::wstring GetWriter() const { return m_Writer; }
		std::wstring GetCoAuthor() const { return m_CoAuthor; }
		SongImpl(std::wstring writer, std::wstring coauthor,
const Lyrics* const lyrics, int yearofrelease)
			:		m_Writer{ writer }, m_CoAuthor{ coauthor },
m_YearOfRelease{ yearofrelease },
m_Lyrics(const_cast<Lyrics*>(lyrics))
		{}
	  S		ongImpl(std::wstring writer, std::wstring coauthor,
std::shared_ptr<Lyrics> lyrics, int yearofrelease)
			:		m_Writer{ writer }, m_CoAuthor{ coauthor },
m_YearOfRelease{ yearofrelease }, m_Lyrics(lyrics)
		{}
	  S		ongImpl(const SongImpl& other) :m_Writer{ other.m_Writer
}, m_CoAuthor{other.m_CoAuthor},
			m		_Lyrics{other.m_Lyrics}
		{}
	};
}
```

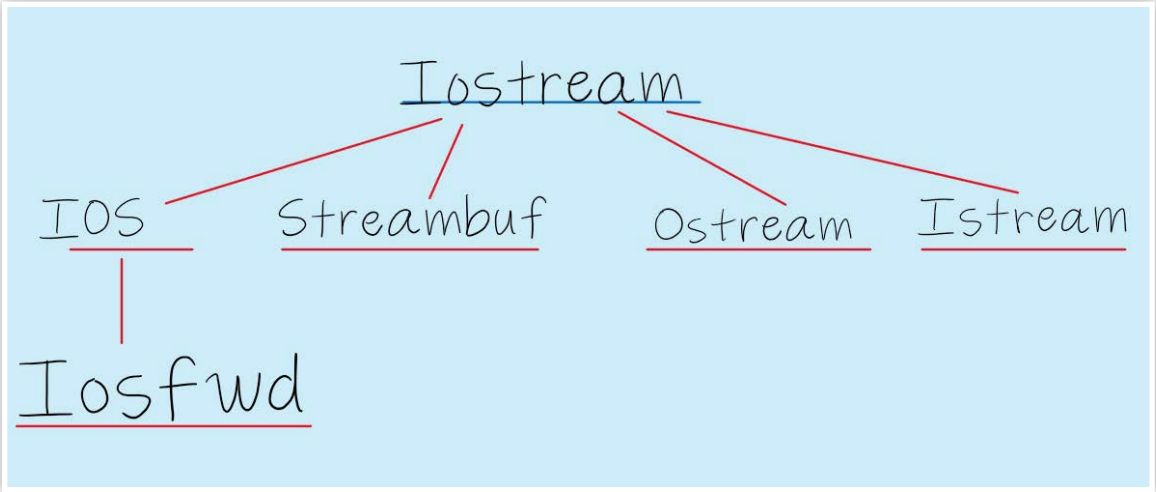
内部实施可以更改。
Client 不需要重新编译。

前向声明

如果你密切关注了 pImpl idiom 用法说明，你也会了解前向声明的要点。在 ShinyNewString 的头文件中，应该只有 Simpl 的前向声明（这是实现类），而不是 #include 整个 Simpl 头文件。

前向声明将类和结构输入头文件中，意味着你只需要在使用这些类的文件中包含相关的头文件，避免了在其他头文件中重复包含头文件，从而减少了编译时间。

仅供参考，这是当你使用 #inlcude＜iostream＞时包含的内容。



为了缩短编译时间，尽量在头文件中使用前向声明，减少包含其他头文件。下面我们重点介绍一个实例，我们在运行示例中使用了前向声明。

```
namespace SongLibrary
{
      class Lyrics;
      class SongImpl
      {
      private:
            std::wstring m_Writer;
            std::wstring m_CoAuthor;
            std::shared_ptr<Lyrics> m_Lyrics;
            int m_YearOfRelease;
      public:
```

前向声明
仅在头文件种使用一个指针。

预编译头文件

预编译头文件是从已解析和预处理的 C 或 C++ 头文件生成的二进制文件。在预编译头文件中，原始文件中的宏和声明都经过排序，从而加快了编译速度。编译期间，编译器检查头文件的修改时间戳是否晚于预编译头文件。如果是，就执行同步以重新创建预编译头文件。

使用预编译头可以将编译时间减少 6 倍。但请记住，在分布式构建过程中，预编译头文件并不总会成功，因为它不是通过调用多个编译进程来并行构建多个单元，而是通过预编译头文件聚合单元，从而防止多个单元编译中断。

请记住，如果预编译头文件的内容需要频繁更改，那么它的优势将不复存在。

在我们的运行示例中，我们已经提到了 pch.h 和 pch.cpp。这是预编译头文件和 cpp 文件。pch.h 包含：

Pch.h (VS 自动生成)

```
// pch.h: This is a precompiled header file.
// Files listed below are compiled only once, improving build
performance for future builds.
// This also affects IntelliSense performance, including code
completion and many code browsing features.
// However, files listed here are ALL re-compiled if any one of them
is updated between builds.
// Do not add files here that you will be updating frequently as
this negates the performance advantage.

#ifndef PCH_H
#define PCH_H

// add headers that you want to pre-compile here
#include "framework.h"

#endif //PCH_H
```

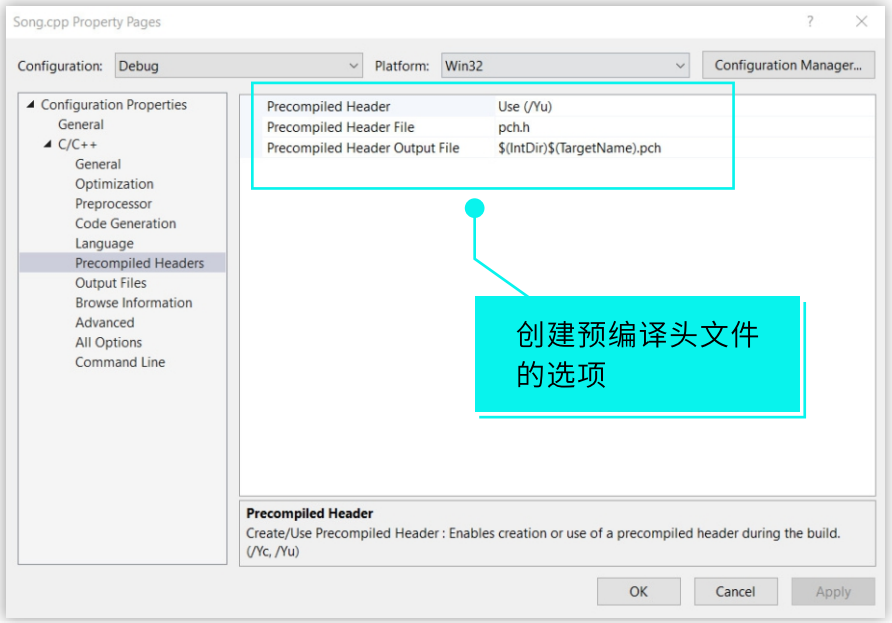
Visual Studio 对预编译头文件的注释很强大。

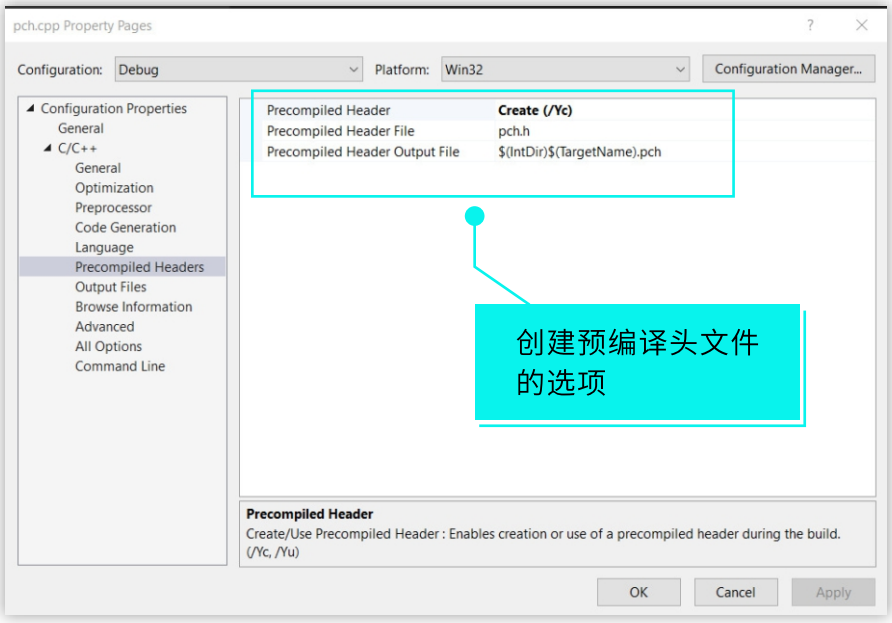
The CPP file pch.cpp contains:

Pch.cpp

```
// pch.cpp: source file corresponding to the pre-compiled header

#include "pch.h"

// When you are using pre-compiled headers, this source file is
necessary for compilation to succeed.
```

更重要的是 Visual Studio 设置了使用 CPP 创建预编译头文件的选项。





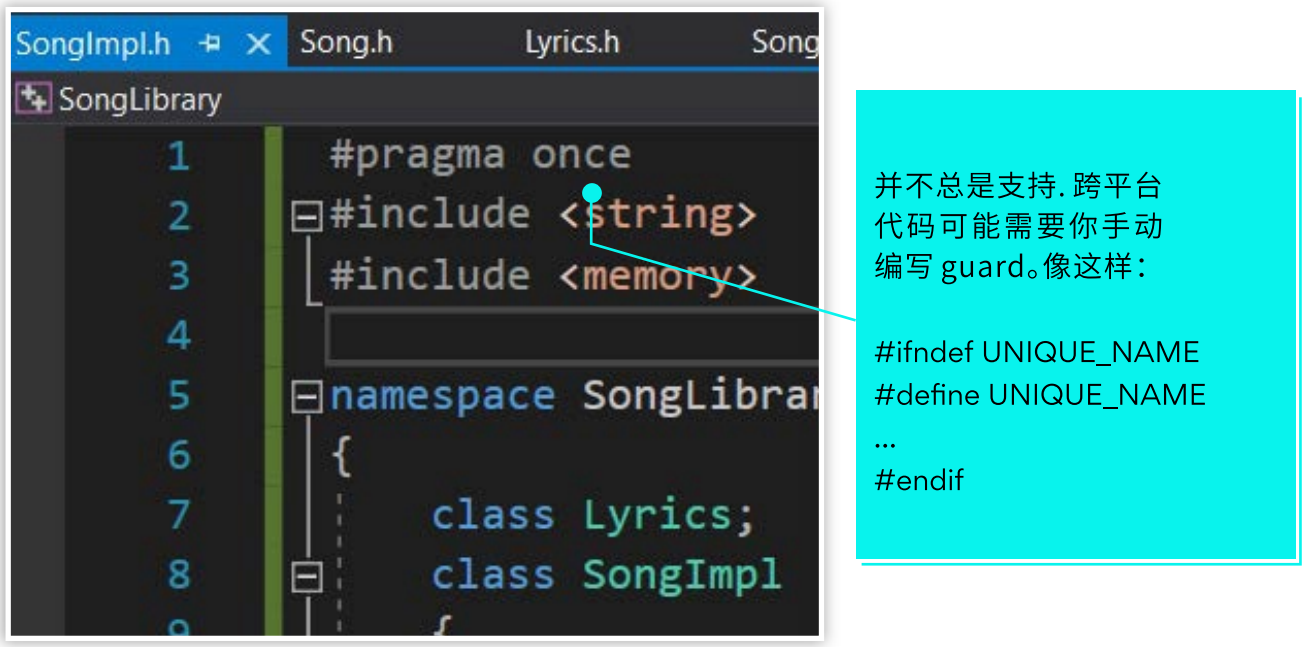
使用 include guard

通过使用 include guard 可以防止在单元编译过程中重复包含头文件。在大多数遵循编码标准（例如，谷歌的编码标准）的项目中，所有标题都必须有 ##define guards，以防止多重包含。

开发者并不总是能成功地想出独一无二的 header guards 名称，这会降低代码的准确度。现代编译器提供了一个 #pragma once 宏，允许编译器在内部为头文件 guard 选择一个唯一的名称。我们建议尽可能使用 #pragma once。

简单地说，防止一个头文件被多次包含，将提高编译时间。因此，请认真遵循这一建议，你会看到 C++ 构建的编译时间将显著减少。

在我们的运行示例中，我们始终使用 #pragma once 作为头文件，因为我们的代码是针对微软 Visual Studio 编译器的。如果你的代码是跨平台的，并且任何平台编译器都不支持 #pragma 指令，那么最好手动创建头文件 guard。



单一编译单元

这种方法颇具争议，但我们仍然看到这种做法可以减少 C++ 构建的编译时间。尽管我们不建议这样做，因为它违背了编译单元的模块化性质，并且对小的增量构建也会有很大的影响。

在这种减少构建时间的方法中，将多个编译单元（CPP 文件）组合在一起，变成一个单一但更大的文件。这提高了构建时间，因为无需重复解析存储与不同 CPP 中的头文件。

使用这种方法时，创建的对象数也减少了，因此减少了链接时间。但使用单一编译单元构建（统一构建）的缺点，是不再可能使用增量构建。这里还值得注意的是，尽管只有头文件的库具有许多优势，但它也增加了 C++ 构建的编译时间。你可以阅读这篇博客深入了解(https://onqtam.com/programming/2018-07-07-unity-builds/)单一构建的利弊。

关闭编译优化器

我们强烈建议不要这样做。编译器比程序员聪明很多，可以大大提高代码的运行时性能。以提高编译时间为幌子来取消编译优化是不可取的。通常，在调试构建期间，好的编译优化会自动发挥作用。这是为了确保调试的二进制文件与源文件匹配。除非你确定自己在做什么，否则我们不建议关闭有效的编译器优化来缩短编译时间。

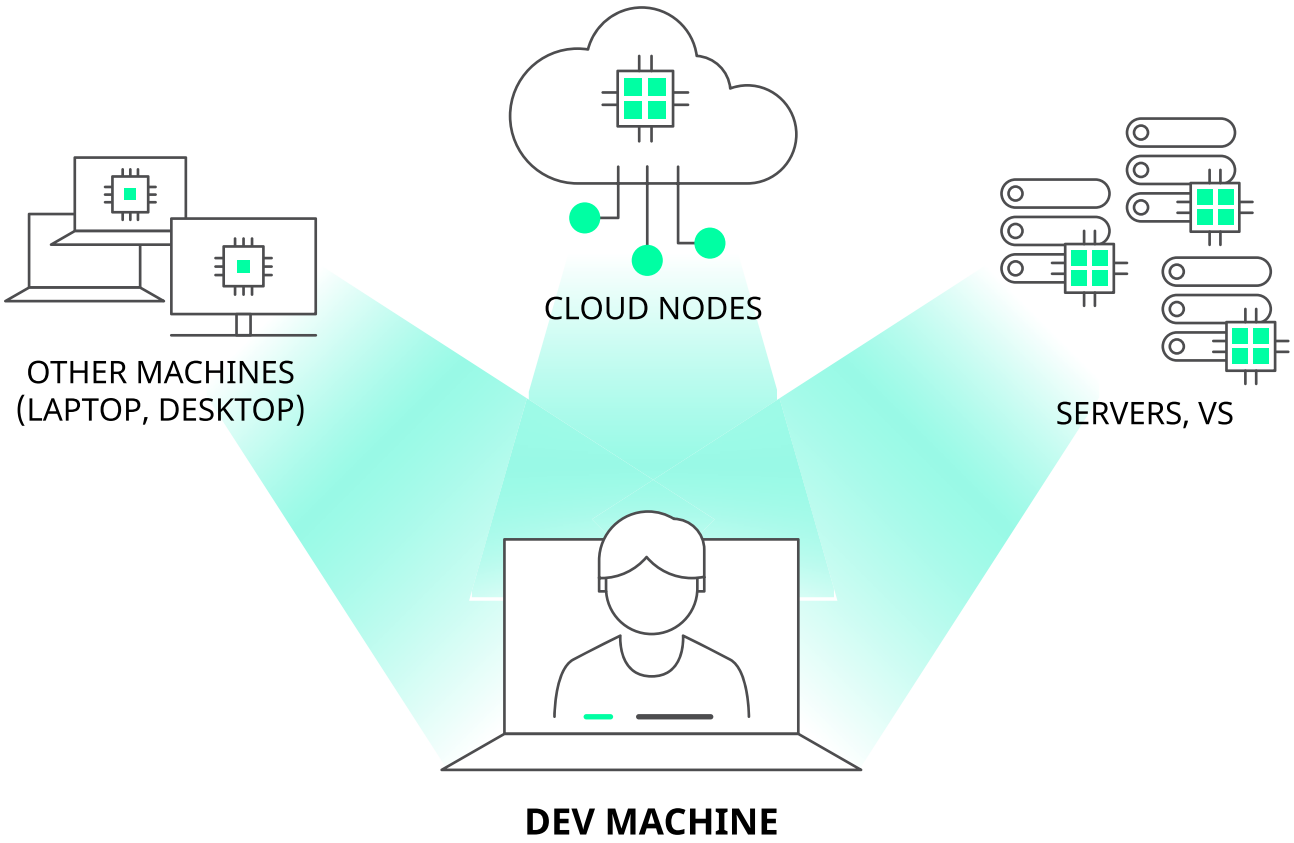
分布式编译
Incredibuild 的解决方案

当然，我们强烈推荐你使用这种方法。 🙂

Incredibuild 就是为解决 C++ 耗时构建难题而生，也因此，我们深受业界推崇，是构建加速领域的世界领导者。

我们的进程虚拟化技术™ 通过网络和云获取闲置 CPU，在远程机器上模拟本地环境，并将每台主机无缝地转换为具有数百甚至数千个内核的超级计算机。这大大提高了构建性能。

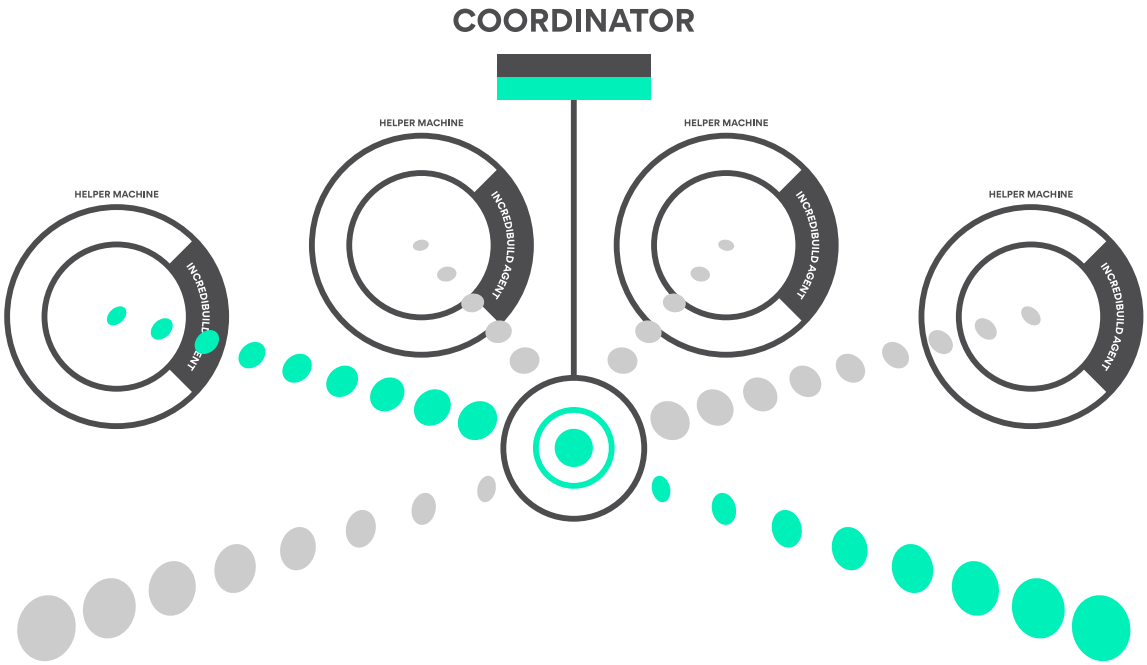
这是我们的运行机制：



安装在每个主机上的 Agent 都连接到一个集中的 coordinator。每个带有 Agent 的主机都可以使用Incredibuild 环境中其他计算机的空闲资源。

在组织环境中，空闲 CPU 的总数很容易高达上千，这些闲置内核的处理能力将有效地帮助加快构建。

从用户的角度来看，这是 Incredibuild 运行时的场景。



主机像是一台拥有数百个内核的超级计算机，编译速度大大加快。

Incredibuild 在远程计算机的安全沙箱上运行进程。进程运行所需的一切，都是由本地主机动态模拟到远程计算机中，我们很安全。

如想了解更多信息，请登录incredibuild.cn，或下载免费的License