Hyperscan读志

一、简介

Hyperscan API 本身由两个主要组件组成：

1、编译

这些函数将一组正则表达式以及标识符和选项标志编译成一个不可变的数据库，以供 Hyperscan 扫描 API 使用。此编译过程执行了大量的分析和优化工作，以便构建一个能够有效匹配给定表达式的数据库。

如果由于任何原因（例如使用不受支持的表达式构造或资源限制溢出）无法将模式构建到数据库中，则模式编译器将返回错误。

编译后的数据库可以序列化和重新定位，以便将其存储到磁盘或在主机之间移动。它们还可以针对特定的平台功能（例如，使用英特尔® 高级矢量扩展 2（英特尔® AVX2）指令）。

2、扫描

一旦创建了 Hyperscan 数据库，它就可以用于扫描内存中的数据。Hyperscan 提供了几种扫描模式，具体取决于要扫描的数据是作为单个连续的块提供的，还是同时分布在内存中的多个块中，还是作为流中的一系列块进行扫描。

匹配结果通过用户提供的回调函数传送给应用程序，该函数在每次匹配时同步调用。

对于给定的数据库，Hyperscan 提供了几项保证：

- 运行时不会发生任何内存分配，但有两个固定大小的分配除外，**对于性能至关重要的应用程序，这两个分配都应该提前完成**：
  - **暂存空间**：扫描时用于存放内部数据的临时内存。暂存空间中的结构在单次扫描调用结束后不再存在。
  - **流状态**：仅在流模式下，需要一些状态空间来存储在每个流的扫描调用之间持续存在的数据。这允许 Hyperscan 跟踪跨越多个数据块的匹配。
- 给定数据库所需的临时空间和流状态（在流模式下）的大小是固定的，并在数据库编译时确定。这意味着应用程序的内存需求是提前知道的，并且如果出于性能原因需要，可以预先分配这些结构。
- 任何已成功由 Hyperscan 编译器编译的模式都可针对任何输入进行扫描。运行时不存在任何内部资源限制或其他限制，这些限制可能会导致扫描调用返回错误。

二、入门

2.1 BUILD

```
1、Build Hyperscan, Depending on the generator used:
cmake --build . – will build everything
make -j<jobs> – use makefiles in parallel
ninja – use Ninja build
MsBuild.exe – use Visual Studio MsBuild


2、Check Hyperscan, Run the Hyperscan unit tests:
bin/unit-hyperscan
```

3、英特尔指令集扩展技术

```
https://www.intel.cn/content/www/cn/zh/support/articles/000005779/processors.html
```

详细说明列在 英特尔®架构指令集扩展编程参考 中。

指令集扩展可包括：

- 单指令多数据 （SIMD）
- 英特尔® Streaming SIMD 扩展（英特尔® SSE、英特尔® SSE2、英特尔® SSE3 和英特尔® SSE4）
- 英特尔® Advanced Vector Extensions（英特尔® AVX、英特尔® AVX2和英特尔® AVX-512）

单击 › 或主题以了解详细信息

全部展开

› **查找英特尔®处理器支持的指令集扩展**

⌄ **流式传输 SIMD 扩展 （SSE）**

SSE 是一种支持单指令多数据的过程或技术。旧款处理器每个指令只处理一个数据元素。SSE 使指令能够处理多个数据元素。它用于 3D 显卡等密集型应用程序，以实现更快的处理速度。SSE 旨在取代 MMX™ 技术。它的数量扩展到了英特尔®处理器的代次，包括 SSE2、SSE3/SSE3S 和 SSE4。每次迭代都带来了新的指令并提高了性能。

查找 采用 SSE 的英特尔®处理器列表。

⌄ **流式传输 SIMD 扩展 2 （SSE2）**

SSE2 通过添加 144 条指令扩展了 MMX 技术和 SSE 技术，可在各种应用中提高性能。以 MMX 技术引入的 SIMD 整数指令从 64 位扩展到 128 位。这使 SIMD 整数运算的有效执行率翻倍。

双精度浮点 SIMD 指令允许以 SIMD 格式同时执行两个浮点运算。这种对双精度运营的支持有助于加快内容创建、金融、工程和科学应用的速度。

增强了原始 SSE 指令，以支持灵活、更高动态的计算功率范围。这是通过支持多种数据类型的算法操作来完成的。示例包括双词和四词。SSE2 指令帮助软件开发人员充分灵活。它们在运行 MPEG-2、MP3 和 3D 显卡等软件时可以实施算法并提供性能增强。

查找 采用 SSE2 的英特尔®处理器列表。

⌄ **流式传输 SIMD 扩展 3 （SSE3）**

基于 90 纳米工艺的英特尔®奔腾® 4 处理器发布，引入了流式传输 SIMD 扩展 3 （SSE3），其中包括比 SSE2 多 13 个 SIMD 指令。这 13 个新指令主要用于改进线程同步和特定的应用区域，如媒体和游戏。

查找 采用 SSE3 的英特尔®处理器列表

⌄ **流式传输 SIMD 扩展 4 （SSE4）**

SSE4 由 54 个指令组成。Penryn提供一个由 47 个指令组成的子集，在英特尔文档中称为 SSE4.1。SSE4.2 是第二个子集，由剩余的七个指令组成，首次在基于 Nehalem 的英特尔® 酷睿™ i7处理器中提供。英特尔将获得开发人员在开发指令集时的反馈。

查找 采用 SSE4.1 的英特尔®处理器列表。

查找 采用 SSE4.2 的英特尔®处理器列表。

⌄ **英特尔® Advanced Vector Extensions（英特尔® AVX和 AVX2）**

英特尔® AVX是一种面向英特尔® SSE 的 256 位指令集扩展，专为浮点 （FP） 密集型应用而设计。英特尔 AVX 由于矢量更宽、新的可扩展语法和丰富的功能而提高性能。英特尔 AVX2于 2013 年发布，扩展了跨浮点和整数数据域的矢量处理能力。这样就可以在各种应用程序上实现更高的性能和更高效的数据管理。例如图像和音频/视频处理、科学模拟、金融分析以及 3D 建模和分析。

查找 采用 AVX 的英特尔®处理器列表。

查找 采用 AVX2 的英特尔®处理器列表。

⌄ **英特尔® Advanced Vector Extensions 512 （英特尔® AVX-512）**

英特尔® AVX-512 一条指令就能处理两倍于英特尔 AVX/AVX2 可处理的数据元件，是英特尔 SSE 功能的四倍。英特尔 AVX-512指令非常重要，因为它们为最苛刻的计算任务提供了更高的性能功能。英特尔 AVX-512指令在设计指令功能时可为编译器提供最高程度的支持。

查找 采用 AVX-512 的英特尔®处理器列表。

4、要求

hyperscan的代码主要是使用c++组成的，所以其中必然可以使用intel所构建的相关c++编译优化（包含编译方式、指令集等）

```
//参考： https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-10/function-annotations-simd-directive-vectorization.html#GUID-38C9CE7E-26DF-4F81-A0AC-B4CEDF284781
```

4.1、硬件

Hyperscan 将在 64 位（Intel® 64 架构）和 32 位（IA-32 架构）模式的 x86 处理器上运行。

Hyperscan 是一个高性能软件库，利用了英特尔架构的最新进展。至少需要支持补充流 SIMD 扩展 3 (SSSE3)，该扩展应在任何现代 x86 处理器上可用。

此外，Hyperscan还可以利用：

> - 英特尔流式 SIMD 扩展 4.2 (SSE4.2)
> - POPCNT 指令
> - 位操作指令（BMI、BMI2）
> - 英特尔高级矢量扩展 2 (英特尔 AVX2)

```
/*
 * ICC and MSVC don't break out POPCNT or BMI/2 as separate pre-def macros
 */
#if defined(__POPCNT__) ||                                                  \
    (defined(__INTEL_COMPILER) && defined(__SSE4_2__)) ||                   \
    (defined(_WIN32) && defined(__AVX__))
#define HAVE_POPCOUNT_INSTR
#endif

#if defined(__BMI__) || (defined(_WIN32) && defined(__AVX2__)) ||           \
    (defined(__INTEL_COMPILER) && defined(__AVX2__))
#define HAVE_BMI
#endif

#if defined(__BMI2__) || (defined(_WIN32) && defined(__AVX2__)) ||          \
    (defined(__INTEL_COMPILER) && defined(__AVX2__))
#define HAVE_BMI2
#endif
```

4.1.1 如何在编译时检测 SSE/SSE2/AVX/AVX2/AVX-512/AVX-128-FMA/KCVI 的可用性

可以查看相应的`lscpu`参数，查看机器的支持指令中是否存在。
或者检索相关的`intel`处理器：

```
    // https://ark.intel.com/content/www/cn/zh/ark/search/featurefilter.html?productType=873&1_Filter-InstructionSetExtensions=3539
```

针对详细的命令。
大多数编译器会自动定义：

```
    __SSE__
    __SSE2__
    __SSE3__
    __SSE4_1__
    __SSE4_2__
    __AVX__
    __AVX2__
```

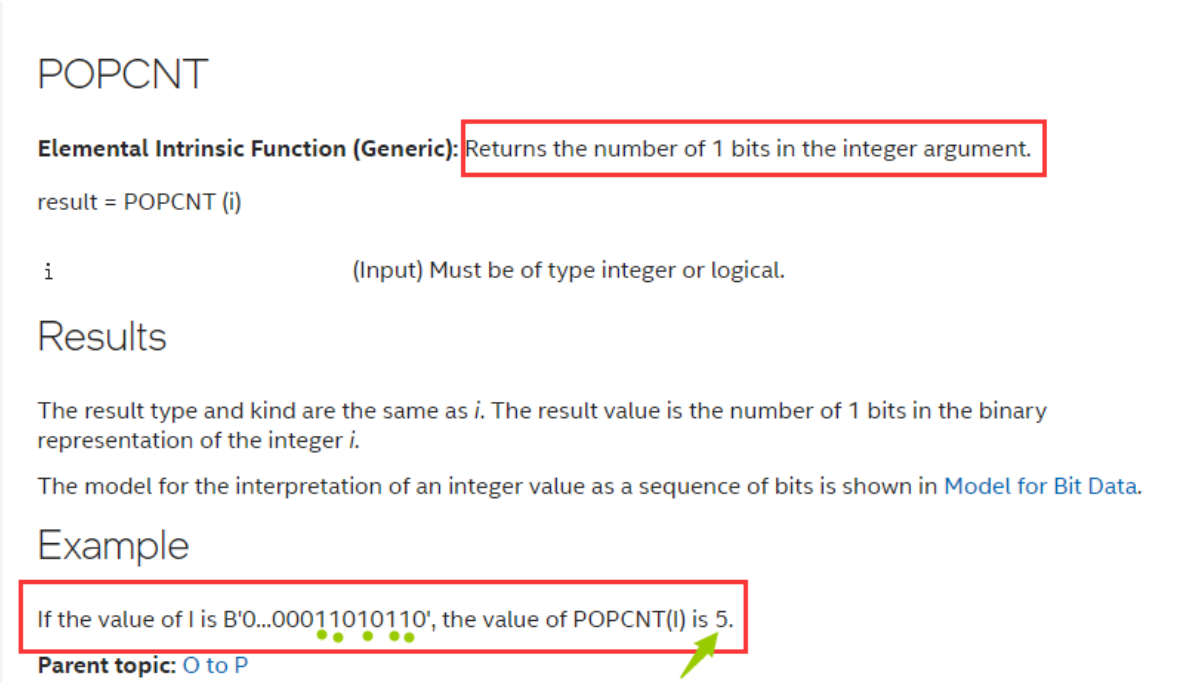
也可以使用 `gcc`（或与 `gcc` 兼容的编译器，如 `clang`）轻松检查这一点，如下所示：

```
    gcc -msse3 -dM -E - < /dev/null | egrep "SSE|AVX" | sort // __SSE3__
    gcc -mavx2 -dM -E - < /dev/null | egrep "SSE|AVX" | sort // __AVX2__
```

或者仅检查特定平台上的默认构建的预定义宏：

```
    gcc -dM -E - < /dev/null | egrep "SSE|AVX" | sort // 查看机器所支持的 SSE|AVX 命令
```

4.1.2 POPCNT 指令作用



POPCNT

Elemental Intrinsic Function (Generic): Returns the number of 1 bits in the integer argument.

result = POPCNT (i)

i	(Input) Must be of type integer or logical.

Results

The result type and kind are the same as *i*. The result value is the number of 1 bits in the binary representation of the integer *i*.

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

Example

If the value of I is B'0...00011010110', the value of POPCNT(I) is 5.

Parent topic: O to P

4.1.3 SIMD 指令作用

```
### 注意，下面的相关描述，只适用于c++的intel编译中，其他GCC中是否适用是不确定的。

/* simd后面可以指定对应的操作参数，具体可以查看
https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-
reference/2021-10/simd.html
    例如 vectorlength (n1[, n2]...)
        vectorlengthfor (data type)
        private (var1[, var2]...)
        firstprivate (var1[, var2]...)
        lastprivate (var1[, var2]...)
        linear (var1:step1 [,var2:step2]...)
        reduction (oper:var1 [,var2]…
        [no]assert
        [no]vecremainder
*/
#pragma simd [clause[ [,] clause]...]   // 强制循环矢量化。
omp simd        // 将循环转换为使用 SIMD 指令并发执行的循环。

    simd pragma 用于引导编译器矢量化更多的循环。使用 simd pragma 进行矢量化是对全自动方法的补充（但不能取代）。
    如果没有明确的 vectorlength() 和 vectorlengthfor() 子句，编译器将使用自己的花费模型(cost model)选择矢量长度。将变量错误分类为 private、firstprivate、lastprivate、linear 和 reduction，或缺乏对变量的适当分类，可能会导致意外后果，如运行失败和/或不正确的结果。
    您最多只能在 private、linear 或 reduction 子句的一个实例中指定一个特定变量。
    如果编译器无法对循环进行矢量化，则会发出警告（使用 assert 子句将其视为错误）。
    如果矢量化器因某种原因不得不停止对循环进行矢量化，SIMD 循环将使用快速浮点模型。
    simd pragma 对该循环执行的矢量化会覆盖为该循环指定的选项 -fp-model (Linux* 和 macOS) 和 /fp (Windows*) 的任何设置。

请注意，simd pragma 可能不会影响所有自动矢量化循环。其中一些循环无法描述 SIMD 向量语义。
```

以下限制适用于 simd pragma：

simd pragma 的可计数循环必须符合 OpenMP 工作共享循环结构的 for-loop 风格。此外，循环控制变量必须是带符号的整数类型。 向量值必须是带符号的 8 位、16 位、32 位或 64 位整数、单精度或双精度浮点数、单精度或双精度复数。

SIMD 循环可能包含另一个循环（for、while、do-while）。这种内部循环不支持 Goto out。支持 Break 和 continue。

注意：内联可以创建这样的内循环，在源代码级可能并不明显。

SIMD 循环无条件执行内存引用。因此，所有地址计算的结果必须是有效的内存地址，即使循环按顺序执行时可能无法访问这些位置。

要禁用可实现更多矢量化的转换，请指定 -vec -no-simd（Linux* 和 macOS）或 /Qvec /Qno-simd（Windows*）选项。

用户强制矢量化（也称 SIMD 矢量化）可以在 #pragma simd 注释循环矢量化失败时断言或不断言错误。默认情况下，simd pragma 设置为 noassert，如果循环矢量化失败，编译器将发出警告。要指示编译器在 #pragma simd 注解循环矢量化失败时断言错误，请在 simd pragma 中添加断言子句。如果编译器未对 simd pragma 注释的循环进行矢量化，该循环将保持其串行语义。

This example shows how to use the simd pragma:

```
void add_floats(float *a, float *b, float *c, float *d, float *e, int n){
 int i;
 #pragma simd
 for (i=0; i<n; i++){
    a[i] = a[i] + b[i] + c[i] + d[i] + e[i];
  }
}
```

在示例中，函数 add_floats() 使用了太多未知指针，编译器的运行时独立性自动检查(例如：根据n值检测对应的a[i]是否有效)优化功能无法启动。程序员可以使用 simd pragma 来强制执行该循环的矢量化，以避免运行时检查的开销。

4.1.4 SSE与AVX区别

SSE (Streaming SIMD Extensions) 和 AVX (Advanced Vector Extensions) 是 Intel 提出的两种不同的 SIMD 指令集扩展，它们在很多方面有显著的区别。以下是它们的详细对比：

SSE (Streaming SIMD Extensions)

推出时间：SSE 最初由 Intel 在 1999 年的 Pentium III 处理器上推出。

指令集版本：包括 SSE、SSE2、SSE3、SSSE3 和 SSE4.1/SSE4.2。

寄存器宽度：SSE 使用 128 位的 XMM 寄存器。

指令集功能：SSE 增加了对浮点运算、整数运算和一些特殊运算（如多媒体处理）的支持。

数据类型支持：支持单精度和双精度浮点数、整数等多种数据类型。

指令数量：随着版本的升级，SSE 指令集数量逐渐增加，涵盖了更广泛的操作。

AVX (Advanced Vector Extensions)

推出时间：AVX 最初由 Intel 在 2011 年的 Sandy Bridge 处理器上推出。

指令集版本：包括 AVX、AVX2、AVX-512 等。

寄存器宽度：AVX 使用 256 位的 YMM 寄存器，而 AVX-512 使用 512 位的 ZMM 寄存器。

指令集功能：

AVX：引入了 256 位浮点运算指令，主要针对单精度和双精度浮点数操作。

AVX2：扩展了 AVX，增加了对整数运算的支持，并引入了更多的指令。

AVX-512：进一步扩展寄存器宽度到 512 位，并引入了更多高级指令，增强了并行计算能力。
数据类型支持：支持单精度和双精度浮点数、整数以及更多的数据类型。
指令数量：相比 SSE，AVX 系列指令更多，功能更强大。

《主要区别》
1、寄存器宽度：
SSE：128 位 XMM 寄存器。
AVX：256 位 YMM 寄存器（AVX2 同样使用 256 位），AVX-512 使用 512 位 ZMM 寄存器。

2、指令集扩展和功能：
SSE：主要增强了浮点运算和多媒体处理能力，逐步扩展支持更多的操作。
AVX：不仅扩展了寄存器宽度，还增加了更多的指令，特别是浮点运算。AVX2 扩展到整数运算，AVX-512 提供了更高级的并行计算能力。

3、性能：
SSE：由于寄存器较窄，处理能力有限，但在其推出时显著提高了处理器的并行计算性能。
AVX：更宽的寄存器和更多的指令使其能够处理更大的数据块，提高了计算效率和性能。

4、应用领域：
SSE：常用于图形处理、多媒体应用、一些科学计算等。
AVX：由于其强大的计算能力，广泛用于高性能计算、科学计算、机器学习、数据压缩等领域。

5、硬件支持：
SSE：几乎所有现代的 Intel 和 AMD 处理器都支持 SSE 指令集。
AVX：较新的 Intel 和 AMD 处理器支持 AVX 和 AVX2，AVX-512 则主要在高端处理器和服务器处理器中支持。

总结：
SSE 和 AVX 都是用于增强处理器并行计算能力的 SIMD 指令集扩展，但它们的设计目标和技术实现有显著差异。AVX 通过更宽的寄存器和更丰富的指令集，显著提升了处理器的性能和计算能力，尤其是在处理大规模数据和高性能计算任务时

1.4 BMI指令

BMI (Bit Manipulation Instruction) 和 BMI2 (Bit Manipulation Instruction Set 2) 是 Intel 和 AMD 在处理器上引入的两套指令集扩展，专门用于提高位操作的效率。这些指令集在处理特定位操作时，可以显著提升性能。以下是关于 BMI 和 BMI2 的详细说明：

BMI (Bit Manipulation Instruction)
推出时间：BMI 指令集在 2011 年由 Intel 推出，首次出现在 Haswell 架构的处理器中，同时 AMD 在其处理器中也有支持。
目标：BMI 指令集旨在提高位操作的效率，这些操作在许多应用中（例如压缩算法、加密、图形处理等）非常常见。
主要指令：
ANDN：按位与非操作，执行 dst = ~a & b。
BEXTR：按位提取，选择并提取特定位段。
BLSI：提取最低设置位，结果为 a & -a。
BLSMSK：生成掩码从最低设置位到最低位的掩码，结果为 (a ^ (a - 1))。
BLSR：重置最低设置位，结果为 a & (a - 1)。
TZCNT：计数尾部的零，等效于 REP BSF 指令，但更有效率。

BMI2 (Bit Manipulation Instruction Set 2)
推出时间：BMI2 指令集作为 BMI 的扩展，由 Intel 在 2013 年推出，并首次出现在 Haswell 架构的处理器中，同时 AMD 也在其处理器中有支持。

目标：BMI2 指令集进一步扩展了位操作指令集，以支持更多复杂的位操作。
主要指令：
BZHI：零高位，清除比指定位置更高的所有位。
PEXT：并行提取，将指定的位提取到连续的低位。
PDEP：并行存储，将低位数据存储到指定的位。
RORX：无影响标志的循环右移。
SARX：无影响标志的算术右移。
SHLX：无影响标志的逻辑左移。
SHRX：无影响标志的逻辑右移。

《主要区别》
1、指令集范围：
BMI：提供了一些基本的位操作指令，主要用于高效地执行常见的位操作。
BMI2：在 BMI 的基础上，扩展了更多复杂的位操作指令，进一步增强了位操作的能力和效率。
2、推出时间：
BMI：2011 年推出。
BMI2：2013 年推出。
3、指令功能：
BMI：主要包括基本的按位操作，如 ANDN、BEXTR、BLSI 等。
BMI2：增加了复杂的并行提取和存储操作（如 PEXT 和 PDEP），以及不影响标志的移位操作（如 RORX、SARX、SHLX 和 SHRX）。

《应用场景》
BMI 和 BMI2 指令集在需要高效位操作的应用中非常有用。例如：
压缩和解压缩算法：如 Huffman 编码、LZ77 压缩等。
加密和解密：如 AES、SHA 算法等。
图形处理：如快速像素操作、图像变换等。
系统软件和低级编程：如操作系统内核、设备驱动程序等。

总结：
BMI 和 BMI2 指令集通过提供高效的位操作指令，显著提高了处理器在特定任务中的性能。BMI 提供了一些基本的位操作指令，而 BMI2 则在其基础上引入了更多复杂的指令，使得处理复杂位操作任务更加高效。这些指令集在压缩、加密、图形处理和系统编程等领域有广泛的应用。

4.2、环境

4.2.1 库需求

In addition, the following software is required for compiling the Hyperscan library:

Dependency	Version	Notes
CMake	>=2.8.11	
Ragel	6.9	
Python	2.7	
Boost	>=1.57	Boost headers required
Pcap	>=0.8	Optional: needed for example code only

4.2.2 Fat Runtime

Hyperscan v4.4 引入了一项功能，即 Hyperscan 库能够为主机处理器调度最合适的运行时代码。此功能称为“Fat Runtime”，因为单个 Hyperscan 库包含不同指令集的运行时代码的多个副本。

> NOTE: Fat Runtime功能仅在 Linux 上可用。Hyperscan 的发布版本将默认启用支持胖运行时的版本。

使用 Fat Runtime 构建库时，Hyperscan 运行时代码将针对这些不同的指令集进行多次编译，并将这些编译后的对象组合成一个库。用户应用程序针对此库的构建方式没有任何变化。

执行应用程序时，会为运行应用程序的机器选择正确的运行时版本。这是通过检查 `CPUID` 是否存在指令集来完成的，然后解析间接函数，以便使用每个 API 函数的正确版本。这对函数调用性能没有影响，因为此检查和解析是在加载二进制文件时由 ELF 加载程序执行的。

如果在不带 的 x86 系统上使用 Hyperscan 库 `SSSE3`，运行时 API 函数将解析为返回的函数 `HS_ARCH_ERROR`，而不是执行可能非法的指令。**应用程序编写者可以使用 API 函数 `hs_valid_platform()` 来确定 Hyperscan 是否支持当前平台。**

从此版本开始，构建的运行时变体以及所需的 CPU 功能如下：

Variant	CPU Feature Flag(s) Required	gcc arch flag
Core 2	`SSSE3`	`-march=core2`
Core i7	`SSE4_2` and `POPCNT`	`-march=corei7`
AVX 2	`AVX2`	`-march=core-avx2`
AVX 512	`AVX512BW` (see note below)	`-march=skylake-avx512`
AVX 512 VBMI	`AVX512VBMI` (see note below)	`-march=icelake-server`

三、编译模式 Compiling

3.1 建立数据库

Hyperscan 编译器 API 接受正则表达式并将其转换为编译的模式数据库，然后可用于扫描数据。

API 提供了三个将正则表达式编译到数据库中的函数：

1. `hs_compile()`：将单个表达式编译成模式数据库。
2. `hs_compile_multi()`：将表达式数组编译成模式数据库。扫描时将同时扫描所有提供的模式，并在匹配时返回用户提供的标识符。
3. `hs_compile_ext_multi()`：编译如上所述的表达式数组，但允许为每个表达式指定扩展参数。

编译允许 Hyperscan 库分析给定的模式并预先确定如何以优化的方式扫描这些模式，而这在运行时的计算成本太高。

在编译表达式时，需要决定是否以流式、块式或矢量式模式使用生成的编译模式：

- **流式模式**：要扫描的目标数据是连续的流，并非所有数据都可以一次获得；数据块按顺序扫描，匹配可能跨越流中的多个块。在流式模式下，每个流都需要一个内存块来存储扫描调用之间的状态。
- **块模式**：目标数据是离散的、连续的块，可以在一次调用中扫描并且不需要保留状态。

- **矢量模式**：目标数据由一系列非连续块组成，这些块可同时使用。与块模式一样，不需要保留状态。

要编译用于流式模式的模式，必须将`mode`的参数`hs_compile()`设置为`HS_MODE_STREAM`；同样，块模式需要使用`HS_MODE_BLOCK`，而矢量模式需要使用`HS_MODE_VECTORED`。为一种模式（流式、块或矢量）编译的模式数据库只能在该模式下使用。用于生成编译模式数据库的 Hyperscan 版本必须与用于扫描的 Hyperscan 版本匹配。

Hyperscan 为针对特定 CPU 平台的数据库提供支持；有关详细信息，请参阅指令集专业化。

3.2 Compile Pure Literals

```
For example, given an expression written as /bc?/.
```

在上面的字符串匹配中，传统的3.1中三个编译函数，会把/bc?/识别为一个Meta数据（正则表达式）来进行复杂的处理。但如果你只是想要一个单纯的“纯文字”录入编译呢：说它是一个纯文字表达式，意思是这是一个3字节长的字符序列，包含字符`b`，`c`和`?`。

为了处理这类情况：我们新增了两个函数：

```
// 在 v5.2.0 中，Hyperscan为纯文字模式引入了两个新的编译API：

hs_compile_lit() ：将单个纯文字编译到模式数据库中。
hs_compile_lit_multi(): ：将纯文字数组编译到模式数据库中。所有提供的模式将在扫描时同时扫描，当它们匹配时返回用户提供的标识符。
```

这两个API是为目标规则集中包含的所有模式都是纯文字的用例而设计的。用户可以将初始的纯文字内容直接传递到这些API中，而不用担心在模式中写入常规的Meta字符。不再需要任何预处理工作。

Note: 我们不支持带有扩展参数的字面编译API。对于运行时实现，仍然可以使用传统的运行时API来匹配纯文字模式。

3.3 libpcre支持

Hyperscan支持PCRE库（“libpcre”）使用的模式语法，如http://www.pcre.org/所述。但是，并非libpcre中所有可用的构造都受支持。使用不受支持的构造将导致编译错误。

3.3.1 Supported Constructs

用于验证Hyperscan对此语法的解释的PCRE版本为8.41或更高版本。

Hyperscan支持以下正则表达式构造：

```
1、文字字符和字符串，所有libpcre引号和字符转义。

2、字符类，如 .（点）、 [abc] 和 [^abc] ，以及预定义的字符类 \s 、 \d 、 \w 、 \v 和 \h 及其取反对应项（ \S 、 \D 、 \W 、 \V 和 \H ）。

3、POSIX命名字符类 [[:xxx:]] ，否定命名字符类 [[:^xxx:]] 。

4、Unicode字符属性，例如 \p{L} 、 \P{Sc} 、 \p{Greek} 。
```

5、Quantifiers:

a、当应用于任意受支持的子表达式时，支持诸如 ? 、 * 和 + 之类的量词。

b、有界重复限定符（如 {n} 、 {m,n} 、 {n,} ）受支持，但有限制。

（1）对于任意重复的子模式：n 和 m 应该很小或无限大，例如 (a|b){4} 、 (ab?c?d){4,10} 或 (ab(cd)*){6,}

（2）对于单字符宽度子模式，例如 [^\a] 或 . 或 x ，几乎支持所有重复计数，除非重复次数非常大（最大界限）大于 32767)。对于大的有界重复，流状态可能非常大，例如 a.{2000}b 。注意：如果在模式的开头或结尾，尤其是在该模式的 HS_FLAG_SINGLEMATCH 标志打开的情况下，此类子模式可能会简洁得多。

c、支持惰性修饰符（ ? 附加到另一个量词，例如 \w+? ），但会被忽略（因为 Hyperscan 报告所有匹配项）。

6、与 | 符号交替，如 foo|bar 。

7、锚点 ^ 、 $ 、 \A 、 \Z 和 \z 。

8、Option modifiers: 选项修饰符:

这些允许为子模式打开（使用 (?<option>) ）和关闭（使用 (?-<option>) ）行为。支持的选项有：

i ：不区分大小写的匹配，按照 HS_FLAG_CASELESS 。

m ：多行匹配，按照 HS_FLAG_MULTILINE 。

s ：将 . 解释为“任何字符”，按照 HS_FLAG_DOTALL 。

x ：扩展语法，它将忽略模式中的大多数空格，以与 libpcre 的 PCRE_EXTENDED 选项兼容。

// 例如，表达式 foo(?i)bar(?-i)baz 将仅对匹配的 bar 部分启用不区分大小写的匹配。

9、\b 和 \B 零宽度断言（分别为字边界和“非字边界”）。

10、采用 (?# comment) 语法的注释。

10、模式开头的 (*UTF8) 和 (*UCP) 控制动词，用于启用 UTF-8 和 UCP 模式。

Note 笔记:

1、具有大量任意表达式重复次数的有界重复量词（例如 ([a-z]|bc*d|xy?z){1000,5000} ）将在模式编译时导致“模式太大”错误。

2、当前，并非所有模式都可以使用 HS_FLAG_SOM_LEFTMOST 标志成功编译，该标志启用：启用的匹配可以报告的匹配到的式子的开始偏移位置。支持该标记的模式是可以使用 Hyperscan 成功编译的模式的子集；特别是，许多可以（未启用匹配开始标记的有界重复形式)而通过Hyperscan 编译，但在启用该标记后就无法编译。对于所有3种模式，启用此行为可能会降低性能。并且特别地，它可能增加流式传输模式中的流状态要求。

3.3.2 Unsupported Constructs

具体的语法定义，可以查看perl regex：https://perldoc.perl.org/perlre#Capture-groups

1、反向引用和捕获子表达式。
2、部分零宽断言。
3、子程序引用和递归模式。Subroutine references and recursive patterns.
4、条件模式。 Conditional patterns
5、回溯控制动词。 Backtracking control verbs.
6、\C “single-byte”指令（中断UTF-8序列）。
7、\R 新行匹配。
8、\K 开始匹配重置指令。
9、标注和嵌入代码。 Callouts and embedded code.
10、原子分组和所有格量词。 Atomic grouping and possessive quantifiers.

正则名称	含义	作用	示例
Backreferences and capturing sub-expressions	反向引用和捕获子表达式	捕获匹配的子串，并在后续匹配中引用它们	`/(.)\1/` 匹配重复的字符，如 "aa" 或 "bb"
Arbitrary zero-width assertions	任意零宽断言	断言条件，零宽度不消耗字符	`/(?<=@)\w+/` 匹配 "@" 后的单词，如 "@user" 中的 "user"
Subroutine references	子程序引用	定义并调用子程序	`/(?<name>(?:\d+))(?(1)\d+)/` 匹配一个数字后跟另一个数字，取决于是否匹配了第一个数字
Recursive patterns	递归模式	处理嵌套的模式	下述
Conditional patterns	条件模式	根据前面的匹配条件选择匹配方式	下述
Backtracking control verbs	回溯控制动词	控制正则表达式的回溯行为	`/(?{0})/` 禁用回溯（示例简化，实际用法复杂）
The \C “single-byte” directive (which breaks UTF-8 sequences)	\C 单字节指令（打断 UTF-8 序列）	处理单字节字符，不考虑 UTF-8 编码	`/\C{2}/` 匹配两个字节的字符，忽略 UTF-8 编码
The \R newline match	\R 换行符匹配	匹配所有类型的换行符	`/\R/` 匹配换行符，包括 `\n`、`\r` 和 `\r\n`
The \K start of match reset directive	\K 匹配重置指令	从匹配位置开始重置，不包括之前的匹配	`/(?<=\d)\K\w+/` 匹配数字后的单词，但不包括数字本身
Callouts and embedded code	调用和嵌入代码	在正则表达式中执行 Perl 代码	`/(?{ print "Matched: $&" })/` 在匹配时打印信息
Atomic grouping and possessive quantifiers	原子分组和贪婪量词	原子分组避免回溯，贪婪量词不允许回溯	`/(?>a*)b/` 匹配任意个 "a" 后跟 "b"，"a" 不回溯

Recursive Patterns（递归模式）

示例：

```
/(?<bracket>\((?:[^()]*|(?&bracket))*\))/
```

说明：

- `(?<bracket>...)`: 这是一个命名捕获组，定义了一个子模式 `bracket`。该模式可以递归地调用自己。
- `\(` 和 `\)`: 匹配一对括号。
- `(?:[^()]*|(?&bracket))*`

  : 这是一个非捕获组，其中包括两个部分：

  - `[^()]\*`: 匹配不包含括号的字符。
  - `(?&bracket)`: 递归引用名为 `bracket` 的子模式，允许在括号内嵌套其他括号。

这个正则表达式可以匹配任意深度的括号嵌套。例如：

- 对于字符串 `"(a(b)c(d))"`，整个字符串会被成功匹配，因为它包含了任意深度的嵌套括号。
-

Conditional Patterns（条件模式）

示例：

```
/(a)?(?(1)b|c)/
```

说明：

- `(a)?`: 匹配一个可选的 `a` 字符，并将其捕获为组 1。`a` 可能出现，也可能不出现。
- `(?(1)b|c)`

  : 条件模式：

  - `(?(1)b|c)`: 这是一个条件匹配，如果组 1 存在（即匹配到了 `a`），则匹配 `b`。如果组 1 不存在（即没有匹配到 `a`），则匹配 `c`。

这个正则表达式根据是否匹配到 `a` 来决定匹配 `b` 还是 `c`。例如：

- 对于字符串 `"a"`，正则表达式会匹配 `b`，结果是 `"b"`。
- 对于字符串 `"b"`，正则表达式会匹配 `c`，结果是 `"c"`。

zero-width assertions （零宽断言）

零宽断言用于匹配某个位置条件，而不匹配实际的字符。这意味着它们检查字符串中字符的位置，而不是字符本身。由于它们不消耗任何字符宽度，所以叫做“零宽断言”。

常见的零宽断言

1. **单词边界 (`\b`)**:
   - 匹配一个单词的开始或结束位置。
   - 示例：`\bword\b` 匹配 `"word"` 在单词边界上的出现，例如 `"word"` 但不会匹配 `"wording"`。
2. **非单词边界 (`\B`)**:
   - 匹配不在单词边界上的位置。
   - 示例：`\Bword\B` 匹配 `"word"` 但不会匹配 `" word "` 或 `"wording"`。
3. **行开始 (`^`)**:
   - 匹配行的开始位置。
   - 示例：`^word` 匹配以 `"word"` 开头的行。
4. **行结束 (`$`)**:
   - 匹配行的结束位置。

- 示例：`word$` 匹配以 `"word"` 结尾的行。

5. **正向前瞻（Positive Lookahead，`(?=...)`）**：
   - 确保当前位置后面跟着指定的子模式，但不包含这些字符。
   - 示例：`a(?=b)` 匹配 `"a"` 只有在后面跟着 `"b"` 的时候。
6. **负向前瞻（Negative Lookahead，`(?!...)`）**：
   - 确保当前位置后面不跟着指定的子模式。
   - 示例：`a(?!b)` 匹配 `"a"` 只有在后面不跟着 `"b"` 的时候。
7. **正向后顾（Positive Lookbehind，`(?<=...)`）**：
   - 确保当前位置前面有指定的子模式。
   - 示例：`(?<=b)a` 匹配 `"a"` 只有在前面跟着 `"b"` 的时候。
8. **负向后顾（Negative Lookbehind，`(?<!...)`）**：
   - 确保当前位置前面不跟着指定的子模式。
   - 示例：`(?<!b)a` 匹配 `"a"` 只有在前面不跟着 `"b"` 的时候。

为什么叫"零宽断言"

- **"零宽"**：这些断言不消耗任何字符，它们仅仅检查字符的位置。例如，`^` 和 `$` 检查行的开始和结束位置，而不是匹配实际的字符。`(?=...)` 和 `(?!...)` 则检查当前位置后面的内容，但不包括这些内容本身。

3.4 Semantics 语义

虽然Hyperscan遵循libpcre语法，但它提供了不同的语义。与libpcre语义的主要偏离是由流和多个同时模式匹配的需求引起的。

1、多模式匹配

Hyperscan允许同时报告多个模式的匹配。这并不等同于通过以下方式分离模式：|在libpcre中，它计算从左到右的变化。

2. Lack of Ordering（无序匹配）

说明：

Hyperscan 在处理多个匹配时，并不保证这些匹配的结果是按某种特定顺序排列的。也就是说，在一个扫描过程中，多个匹配可能会以任意顺序出现。这是因为 Hyperscan 的设计重点是处理高效的模式匹配，而不是保证匹配的顺序。

- **匹配的范围**：尽管匹配的结果可能无序，但是所有的匹配都会在当前扫描的范围内。例如，如果你在处理一个文本片段，所有匹配的模式都会出现在这个文本片段内。
- **应用场景**：这种无序的匹配适用于不依赖于匹配顺序的场景，例如在网络流量分析中，你可能只关心是否有某些模式出现，而不关心它们出现的顺序。

3. End Offsets Only（仅报告结束偏移）

说明：

Hyperscan 默认只报告匹配的结束偏移，即模式匹配在输入文本中的结束位置。这个行为对于许多应用场景是足够的，但在某些情况下，可能需要报告匹配的开始偏移。

- **开始偏移**：如果你需要匹配的开始位置，你可以在模式编译时使用特定的标志启用对开始偏移的报告。Hyperscan 提供了这样的选项，但这需要在编译模式时进行设置。

- **编译时标志**：在编译模式时，你可以使用标志来指定是否需要报告开始偏移。这些标志会影响 Hyperscan 在生成模式时的行为，从而使其能够报告开始偏移。

示例配置：

假设你正在编译一个正则表达式模式并希望获得匹配的开始和结束位置，你可以在编译时设置相应的标志。例如：

```
hs_compile_expression("pattern", HS_FLAG_SOM_START, HS_MODE_BLOCK, &db, NULL);
```

在这里，`HS_FLAG_SOM_START` 标志会使 Hyperscan 在匹配时报告开始偏移。

4、“All matches”reported

```
说明：
    扫描/foo.* bar/用于匹配 fooxyzbarbar. 将从Hyperscan返回两个匹配-在对应于fooxyzbar和fooxyzbarbar端点的点处。相反，libpcre语义在默认情况下只报告fooxyzbarbar上的一个匹配（贪婪语义），或者如果打开非贪婪语义，则报告fooxyzbar上的一个匹配。
    这意味着在贪婪和非贪婪语义之间切换在Hyperscan中是无效操作。
```

libpcre量词语义：量词的语义定义了模式匹配中量词（如 `*`、`+`、`?`）的行为

支持libpcre量词语义，同时在流匹配发生时准确地报告它们是不可能的。例如，考虑上面的模式，`/foo.* bar/`，在流模式下，针对以下流（顺序扫描的三个块）：

block 1 块1	block 2 块2	block 3 块3
`fooxyzbar`	`baz`	`qbar`

libpcre中的*贪婪*重复，它必须尽可能地匹配，而不会导致模式的其余部分失败。然而，在流式传输模式中，这将需要了解流中超出正在扫描的当前块的数据（未来的数据块）。

在这个例子中，第一块中偏移量9处的匹配（fooxyzbar）只有在后续块中没有`匹配命中`时才是正确的匹配（在libpcre语义下），如块3中那样-这将构成模式的更好匹配（fooxyzbarbazqbar）。

在流模式中，处理上述情况变得复杂，因为：

1. **局部视野**：流模式通常只对当前块的内容可见，而对未来块的内容不可见。这使得贪婪量词的匹配变得复杂，因为贪婪量词可能需要查看后续块的数据，以确保能够找到最佳匹配。
2. **后续块中的数据**：在这个例子中，**Block 3** 中的 `qbar` 可能会被认为是一个更好的匹配，因为它的 `bar` 紧随其后，没有 `.*` 的干扰。**Block 3** 中的匹配 `qbar` 比 **Block 1** 中的 `fooxyzbar` 更符合 `foo.*bar` 模式的语义。
3. **保留匹配结果**：由此，在流模式匹配中，保留上次block的匹配结果就变得尤为重要，匹配逻辑也会由此变得更加复杂。

3.5 SOM(start of match)

`HS_FLAG_SOM_LEFTMOST` 是 Hyperscan 编译标志之一，用于控制模式匹配时是否报告最左侧的匹配起始位置。以下是该标志的详细说明：

标志作用

- 启用左侧匹配起始位置报告
    - 默认情况下，Hyperscan 仅报告匹配的结束位置，而不报告开始位置。

- 启用 `HS_FLAG_SOM_LEFTMOST` 标志后，Hyperscan 会在报告匹配时，同时报告匹配的最左侧起始位置。这意味着即使模式在多个位置匹配，Hyperscan 会报告最早的那个匹配开始位置。

详细说明

1. **左侧起始位置**：
   - **默认行为**：Hyperscan 默认只报告匹配的结束偏移量。例如，对于模式 `/foo.*bar/`，如果在输入流中找到匹配，Hyperscan 会返回匹配的结束位置，而不返回匹配的开始位置。
   - **启用 `HS_FLAG_SOM_LEFTMOST`**：使用该标志后，Hyperscan 在返回匹配结果时，会报告匹配的最左侧的起始位置。这对于需要精确知道匹配开始位置的应用非常有用。
2. **性能影响**：
   - **性能降低**：启用 `HS_FLAG_SOM_LEFTMOST` 可能会导致性能下降。这是因为计算匹配的最左侧起始位置需要额外的处理，这可能会增加匹配操作的复杂度。
   - **流模式影响**：在流模式下，启用此标志可能会显著增加流状态的需求。流模式是逐块扫描输入数据的，如果需要跟踪每个块中的最左侧匹配位置，这可能会增加内存使用和计算开销。

使用SOM标志需要进行一些权衡和限制：

减少模式支持：对于许多模式，跟踪SOM是复杂的，并且可能导致Hyperscan无法编译模式并出现“模式太大”错误，即使该模式在正常操作中受支持。

增加流状态：在扫描时，需要状态空间来跟踪潜在的SOM偏移，并且这必须以流模式存储在持久流状态中。因此，SOM通常将增加匹配模式所需的流状态。

性能开销：类似地，跟踪SOM通常会带来性能开销。

不兼容的功能：某些其他Hyperscan模式标志（如HS_FLAG_SINGLEMATCH和HS_FLAG_PREFILTER）不能与SOM结合使用。将它们与HS_FLAG_SOM_LEFTMOST一起编译将导致编译错误。

在流模式下，SOM提供的精度可以通过SOM范围标志来控制。这些参数指示Hyperscan在结束偏移的特定距离内提供准确的SOM信息，否则返回特殊的开始偏移 **HS_OFFSET_PAST_HORIZON。适量的减小SOM范围内，通常会减少给定数据库所需的流状态，降低匹配压力甚至时编译压力。**

```
Note:
    ##  在流式传输模式中，针对匹配返回的开始偏移可以指的是流中在“当前块被扫描之前的偏移点”。Hyperscan不提供访问早期块的工具;如果调用应用程序需要检查历史数据，则它必须自己存储这些数据。
```

3.6 Extended Parameters 扩展参数

在某些情况下，需要对模式的匹配行为进行更多的控制，而不是使用正则表达式语法轻松指定。对于这些场景，Hyperscan提供了 `hs_compile_ext_multi()` 函数，该函数允许在每个模式的基础上设置一组“扩展参数”。

`hs_expr_ext_t` 结构体

该结构体用于指定扩展的匹配参数，以便在模式编译时可以精确控制匹配行为。结构体的字段包括：

1. `flags`：指定哪些其他字段在结构体中被使用的标志。通过设置适当的标志，可以启用或禁用某些字段的作用。
2. `min_offset`：指定数据流中匹配成功的模式的最小结束偏移量。也就是说，模式的匹配必须在此偏移量之后才能被视为成功。这个参数用于控制模式开始匹配的起始位置。
3. `max_offset`：指定数据流中匹配成功的模式的最大结束偏移量。即模式的匹配必须在此偏移量之前结束才能被视为成功。这个参数用于限制模式的匹配结束位置。
4. `min_length`：指定模式匹配的最小长度（从开始到结束）。模式的匹配结果必须至少达到这个长度才被认为是成功的。这个参数用于控制模式匹配的最小长度。
5. `edit_distance`：允许模式在给定的 Levenshtein 距离内进行匹配。Levenshtein 距离表示从一个字符串到另一个字符串的最小编辑操作次数（插入、删除、替换）。这个参数用于支持模糊匹配。
6. `hamming_distance`：允许模式在给定的 Hamming 距离内进行匹配。Hamming 距离表示两个相同长度的字符串之间不同字符的个数。这个参数也用于支持模糊匹配。

假设你要编译一个模式 `foo`，并希望设置一些扩展的匹配条件：

```
hs_expr_ext_t ext_params = {
    .flags = HS_EXT_FLAG_MIN_OFFSET | HS_EXT_FLAG_MAX_OFFSET |
HS_EXT_FLAG_MIN_LENGTH,
    .min_offset = 5,
    .max_offset = 50,
    .min_length = 3,
    .edit_distance = 1,
    .hamming_distance = 2
};

// 编译模式时传递扩展参数
hs_compile_ext_multi(patterns, num_patterns, HS_MODE_BLOCK, &db, &ext_params);
```

在此示例中：

- `min_offset = 5`：模式 `foo` 必须在数据流的第 5 个字符后才被视为匹配成功。
- `max_offset = 50`：模式 `foo` 的匹配必须在第 50 个字符之前结束。
- `min_length = 3`：模式 `foo` 的匹配结果至少要有 3 个字符长。
- `edit_distance = 1`：允许匹配的模式在最多 1 次编辑操作（插入、删除、替换）内匹配目标模式。
- `hamming_distance = 2`：允许模式在最多 2 个不同字符的 Hamming 距离内匹配目标模式。

例如:

1、模式`/foo.* bar/`当给定`min_offset`为10和`max_offset`为15时，在针对foobar或foo 0123456789 bar扫描时不会产生匹配```，但会针对数据流`foo 0123 bar`或`foo 0123456 bar产生匹配`。

2、同样，当给定`edit_distance`为2时，模式`/foobar/`在与`foobar`、`f00 bar`、`fooba`、`fobr`、`fo_baz`、`fooobar`以及编辑距离为2（由**Levenshtein**距离定义）内的任何其他内容进行扫描时将产生匹配。

3、当相同的模式`/foobar/`的`hamming_distance`为2时，它将在扫描`foobar`、`boofar`、`f00 bar`以及从原始模式中替换最多两个字符的任何其他字符时产生匹配。有关更多详细信息，请参阅近似匹配部分。

3.6.1 hamming 距离

Hamming 距离 是一种用于衡量两个相同长度字符串之间差异的度量。具体来说，Hamming 距离定义为两个字符串中相同位置上不同字符的数量。它用于评估两个字符串之间的相似性，特别是在需要比较或匹配相同长度的字符串时。

详细解释

- **定义**：两个字符串的 Hamming 距离是它们之间不同字符的数量。对于两个长度相同的字符串 `s1` 和 `s2`，Hamming 距离可以通过比较每个字符的位置来计算。
- **公式**：Hamming 距离可以通过如下方式计算：

$$d_H(s1, s2) = \sum_{i=1}^{n} [s1_i \neq s2_i]$$

  其中，`n` 是字符串的长度，`[s1_i != s2_i]` 是一个指示函数，如果 `s1_i` 和 `s2_i` 不相等，则其值为 1，否则为 0。

示例

假设有两个字符串：

- **字符串1**：`"karolin"`
- **字符串2**：`"kathrin"`

它们的 Hamming 距离计算如下：

字符位置	字符1	字符2
1	k	k
2	a	a
3	r	t
4	o	h
5	l	r
6	i	i
7	n	n

不同的字符位置为 3、4 和 5，因此 Hamming 距离为 3。

应用

- **错误检测和纠正**：Hamming 距离用于错误检测和纠正算法中，例如 Hamming 码。
- **基因序列比对**：在生物信息学中，用于比较 DNA、RNA 或蛋白质序列。
- **模式匹配**：在正则表达式或搜索算法中，用于模糊匹配时，允许在给定的 Hamming 距离内匹配模式。

3.6.2 Levenshtein 距离

也称为编辑距离，是一种衡量两个字符串之间的相似性的度量。它定义为将一个字符串转换成另一个字符串所需的最少编辑操作次数，这些编辑操作包括插入、删除或替换字符。Levenshtein 距离可以用来评估字符串之间的相似度和差异性。

详细定义

- **编辑操作**：Levenshtein 距离的计算基于以下三种基本操作：
  - **插入**：在字符串中插入一个字符。
  - **删除**：从字符串中删除一个字符。
  - **替换**：将一个字符替换为另一个字符。
- **公式**：

  dL(s1,s2)=最小编辑操作数

  其中，`s1` 和 `s2` 是两个字符串。

示例

假设有两个字符串：

- **字符串1**：`"kitten"`
- **字符串2**：`"sitting"`

要将 `"kitten"` 转换为 `"sitting"`，需要的操作如下：

1. **替换** `k` → `s`（`"kitten"` → `"sitten"`）
2. **替换** `e` → `i`（`"sitten"` → `"sittin"`）
3. **插入** `g`（`"sittin"` → `"sitting"`）

总共需要 3 次操作，所以 `"kitten"` 和 `"sitting"` 之间的 Levenshtein 距离是 3。

Levenshtein 距离的计算方法

计算 Levenshtein 距离通常使用动态规划算法，以下是基本步骤：

1. **初始化**：创建一个矩阵 `D`，其中 `D[i][j]` 表示将前 `i` 个字符的字符串转换为前 `j` 个字符的字符串所需的最少操作数。
2. **填充矩阵**：
   - `D[0][0]` = 0（空字符串到空字符串的距离为 0）
   - 第一行和第一列分别表示从空字符串到一个非空字符串的距离（即插入或删除操作）。
3. **递归关系**：

$$D[i][j] = \min \begin{cases} D[i-1][j] + 1 & (\text{删除}) \\ D[i][j-1] + 1 & (\text{插入}) \\ D[i-1][j-1] + \text{cost} & (\text{替换，cost 为 0 如果字符} \end{cases}$$

4. **结果**：`D[m][n]` 是将长度为 `m` 的字符串转换为长度为 `n` 的字符串所需的最少操作数。

应用

- **拼写检查**：用于计算词汇之间的相似性。
- **文本比对**：用于相似文档或文本的比较。
- **生物信息学**：用于比较 DNA、RNA 或蛋白质序列。
- **模糊匹配**：在搜索引擎中用于处理拼写错误或近似匹配。

3.6.3 hamming 与 Levenshtein 的区别

主要区别

1. **适用范围**：
   - **Hamming 距离**：仅适用于长度相同的字符串或序列。
   - **Levenshtein 距离**：适用于任何长度的字符串，包括长度不同的字符串。
2. **操作类型**：
   - **Hamming 距离**：只考虑字符替换（相同位置的不同字符）。
   - **Levenshtein 距离**：考虑插入、删除和替换操作，处理更复杂的变换。
3. **应用场景**：
   - **Hamming 距离**：常用于固定长度的序列，如编码理论中的错误检测和纠正。
   - **Levenshtein 距离**：常用于字符串处理任务，如拼写检查、模糊匹配和数据清理。

3.7 Prefiltering Mode 预过滤器

Hyperscan提供了一个每模式标志`HS_FLAG_PREFILTER`，它可用于为Hyperscan通常不支持的模式实现预过滤器。

此标志指示Hyperscan**编译此模式的"近似"版本**，以便在预过滤应用程序中使用，即使Hyperscan在正常操作中不支持该模式。

使用此标志时返回的匹配集保证是由非预筛选表达式指定的匹配的超集。（即产生的匹配集包含了"指定匹配的数集"）

如果模式包含Hyperscan**不支持的模式构造**（查看3.3.2）（如零宽度断言、反向引用或条件引用），这些构造将在内部被更广泛的构造替换，这些构造可能更经常匹配。

> 例如：模式 `/(\w+) again \1/` 包含反向引用`\1`。在预过滤模式下，这个模式可以通过将它的后向引用替换为它的所指对象来近似，形成 `/\w+ again \w+/`。

通常，应用程序随后将使用另一个正则表达式匹配器来确认预过滤器匹配，该匹配器可以为模式提供精确匹配。

> 注意
> # 当前不支持将此标志（HS_FLAG_PREFILTER）与匹配开始模式（HS_FLAG_SOM_LEFTMOST标志）结合使用，这将导致模式编译错误。

总结说明：

1. 编译标志：启用预过滤模式
   解释：这个标志指示 Hyperscan 编译一个“近似”的版本的正则表达式，以用于预过滤应用，即使 Hyperscan 在正常操作模式下不支持这个模式。
2. 预过滤模式的匹配结果
   解释：使用此标志时，Hyperscan 编译的模式会返回一个匹配集合，这个集合保证包含了非预过滤表达式指定的所有匹配项。换句话说，预过滤的匹配结果总是比未使用预过滤的模式的匹配结果多。
3. 模式构造的处理
   解释：如果模式包含 Hyperscan 不支持的模式构造（如零宽断言、回溯引用或条件引用），这些构造将在内部被替换为更宽泛的构造，这可能会导致更多的匹配。例如，Hyperscan 可能会将 \b 或 (?<=...) 等构造转换为更简单的模式。
4. 预过滤模式的优化
   解释：在预过滤模式下，Hyperscan 可能会简化一个在编译时会返回“模式过大”错误的模式，或者为了性能优化而简化模式，但仍然保证匹配的集合是“超集”。
5. 后续精确匹配
   解释：预过滤模式的使用一般是为了快速筛选数据，然后再通过另一个正则表达式匹配器（可以提供精确匹配）来确认这些预过滤的匹配。这样可以确保最终的匹配结果是准确的。
6. 不支持的标志组合
   解释：目前，HS_FLAG_PREFILTER 标志与 HS_FLAG_SOM_LEFTMOST 标志的组合是不被支持的。HS_FLAG_SOM_LEFTMOST 是用来指示 Hyperscan 在匹配时返回最左端的匹配起始位置，而这种与预过滤模式一起使用的情况暂时不被支持。

3.8 Instruction Set Specialization

Hyperscan能够利用x86处理器上的几个现代指令集功能来提高扫描性能。

其中一些功能是在构建库时选择的;例如，Hyperscan将在可用的处理器上使用本机 `POPCNT` 指令，并且库已针对主机体系结构进行了优化。

注意
默认情况下，Hyperscan运行时是用-march=native编译器标志构建的，并且（在可能的情况下）将使用主机的C编译器已知的所有指令。

POPCNT 指令是一种高效的 CPU 指令，用于计算二进制数据中 1 的数量。它能够显著提升处理相关计算的速度，特别是在需要处理大量位操作的场景中。

然而，要使用某些指令集功能，Hyperscan必须构建专门的数据库来支持它们。这意味着目标平台必须在模式编译时指定。

Hyperscan编译器API函数都接受一个可选的 `hs_platform_info_t` 参数，该参数描述要构建的数据库的目标平台。如果此参数为NULL，则数据库将以当前主机平台为目标

hs_platform_info_t结构有两个字段：

```
typedef struct hs_platform_info {
    /**
     * Information about the target platform which may be used to guide the
     * optimisation process of the compile.
     *
     * Use of this field does not limit the processors that the resulting
     * database can run on, but may impact the performance of the resulting
     * database.
```

```
     */
    // 这允许应用程序指定有关目标平台的信息，这些信息可用于指导编译的优化过程。使用此字段不会限制结果数据库可以在其上运行的处理器，但可能会影响结果数据库的性能。
    unsigned int tune;

    /**
     * Relevant CPU features available on the target platform
     *
     * This value may be produced by combining HS_CPU_FEATURE_* flags (such as
     * @ref HS_CPU_FEATURES_AVX2). Multiple CPU features may be or'ed together
     * to produce the value.
     */
    // 这允许应用程序指定可能在目标平台上使用的CPU功能的掩码。例如，HS_CPU_FEATURES_AVX2可以指定为英特尔®高级矢量扩展指令集2（英特尔® AVX 2）支持。如果为特定的CPU功能指定了标志，则数据库将无法在没有该功能的CPU上使用。
    unsigned long long cpu_features;

    /**
     * Reserved for future use.
     */
    unsigned long long reserved1;

    /**
     * Reserved for future use.
     */
    unsigned long long reserved2;
} hs_platform_info_t;
```

第一步：填充设置 `hs_platform_info_t platform_info`
第二步：可以使用 `hs_populate_platform（platform_info）`函数构建针对当前主机的 `hs_platform_info_t` 结构。从而可以帮助优化 Hyperscan 编译过程，使其能根据当前硬件特性进行更高效的配置

3.9 Approximate matching

具体查看3.5-3.5.3

1. **编辑距离**定义为Levenshtein距离。也就是说，考虑了三种可能的编辑类型：插入、移除和替换。更正式的描述可以在维基百科上找到。
2. **Hamming 距离**是两个相等长度的字符串之间的位置差。也就是说，它是将一个字符串转换为另一个字符串所需的替换数。当使用汉明距离进行近似匹配时，不存在插入或移除。更正式的描述可以在维基百科上找到。
3. **近似匹配**将匹配给定编辑或Hamming 距离内的所有*语料库*。也就是说，给定一个模式，近似匹配将匹配任何可以编辑的内容，以获得与原始模式完全匹配的语料库。
4. **匹配语义**与语义中描述的完全相同。

权衡利弊:

- 减少模式支持：
    - 对于许多模式，近似匹配是复杂的，并且可能导致Hyperscan无法编译模式并出现“Pattern too large”错误，即使该模式在正常操作中受到支持。

- 此外，有些模式不能近似匹配，因为它们简化为所谓的“空”模式（匹配所有内容的模式）。例如，如果实现了编辑距离为3的模式 `/foo/`，则会减少到匹配零长度缓冲区。这样的模式将导致“模式无法近似匹配”编译错误。汉明距离内的近似匹配不会删除符号，因此不会减少到一个空洞的模式。
- 最后，由于定义匹配行为的固有复杂性，近似匹配实现了正则表达式语法的精简子集。近似匹配不支持UTF-8（和其他多字节字符编码）和字边界（即 `\B`、`\B` 和其他等效结构）。包含不支持的构造的模式将导致“模式无法近似匹配”编译错误。
- 当结合SOM使用近似匹配时，SOM的所有限制也适用。请参阅比赛开始以了解更多详细信息。

- 增加的流状态/字节码大小要求：由于近似匹配字节码固有地比精确匹配更大且更复杂，相应的要求也增加。
- 性能开销：类似地，由于增加的匹配复杂性，以及由于它将产生更多匹配的事实，通常存在与近似匹配相关联的性能成本。

默认情况下，近似匹配始终处于禁用状态，并且可以通过使用[扩展参数3.5]中描述的扩展参数在每个模式的基础上启用

3.10 Logical Combinations

Hyperscan提供了对给定模式集中模式的逻辑组合的支持，具有**三个运算符：** `NOT`，`AND` **和** `OR`**。**

这种组合的逻辑值基于每个表达式在给定偏移量处的匹配状态。**任何表达式的匹配状态都有一个布尔值**：如果表达式还没有匹配，*则为false;*如果表达式已经匹配，*则为true*。NOT则会取反。

为了说明，这里有一个示例组合表达式：

```
((301 OR 302) AND 303) AND (304 OR NOT 305)
// 如果表达式301在偏移量10处匹配，则301的逻辑值为*真*，而其他模式的值为*假*。因此，整个组合的值为*false*
```

In a logical combination expression:

> - The priority of operators are `!` > `&` > `|`. For example:
>
>   `A&B|C` is treated as `(A&B)|C`,
>
>   `A|B&C` is treated as `A|(B&C)`,
>
>   `A&!B` is treated as `A&(!B)`.
> - Extra parentheses are allowed. For example:
>
>   `(A)&!(B)` is the same as `A&!B`,
>
>   `(A&B)|C` is the same as `A&B|C`.
> - Whitespace is ignored.

```
// 我们以  ((301 OR 302) AND 303) AND (304 OR NOT 305)  为例
#include <hs/hs.h>
#include <stdio.h>
#include <string.h>

// 定义逻辑组合的模式:可以进行适量的拆分组合，有利于逻辑处理函数 process_match 的业务编写
const char *patterns[] = {
    "(301|302)&303",      // (301 OR 302) AND 303
```

```
    "(304|!305)"          // 304 OR NOT 305
};
// 如果你的compile时没有标志 HS_FLAG_COMBINATION， 则不能使用`NOT`, `AND`和`OR`。
// 可必须拆分为五个匹配模式
const char *patterns[] = {
    "301",
    "302",
    "303",
    "304",
    "305"
  }
// 要达到 ((301 OR 302) AND 303) AND (304 OR NOT 305) 这样的逻辑关系，就必须在
process_match 函数中自行编写判断
// 但如果你使用了 HS_FLAG_COMBINATION，甚至也可以直接使用一个模式 ((301 OR 302) AND 303)
AND (304 OR NOT 305) 即可。

            |
            |
            |
            |

// 以下是使用 HS_FLAG_COMBINATION，将表达式拆分为两部分的调用示例：

unsigned int ids[] = {1, 2}; // 给每个模式一个唯一的 ID

// 匹配逻辑处理函数
static int process_match(unsigned int id, unsigned long long from, unsigned long
long to, void *ctx) {
    unsigned int *matches = (unsigned int *)ctx;

    // 标记匹配的 ID
    matches[id - 1] = 1;
    if (matches[0] && matches[1]) // ((301 OR 302) AND 303) AND (304 OR NOT 305)
            return HS_SCAN_TERMINATED; // 终止扫描
    return HS_SCAN_CONTINUE; // 继续扫描
}

int main() {
    hs_database_t *database;
    hs_compile_error_t compile_err;
    hs_error_t err;
    hs_scratch_t *scratch;

    // 编译正则表达式，启用逻辑组合标志
    err = hs_compile_multi(
        2, // 模式数量
        patterns,
        ids,
        HS_MODE_BLOCK | HS_FLAG_COMBINATION, // 启用组合模式
        NULL,
        NULL,
        &database,
        &compile_err
    );
    if (err != HS_SUCCESS) {
```

```
        fprintf(stderr, "Compilation failed: %s\n", compile_err.message);
        return 1;
    }

    // 创建预分配的 scratch 区域
    err = hs_alloc_scratch(database, &scratch);
    if (err != HS_SUCCESS) {
        fprintf(stderr, "Failed to allocate scratch space: %s\n",
hs_strerror(err));
        hs_free_database(database);
        return 1;
    }

    // 匹配示例数据
    const char *data = "301 303 304";
    err = hs_scan(database, data, strlen(data), 0, scratch, process_match,
NULL);
    if (err != HS_SUCCESS) {
        fprintf(stderr, "Scan failed: %s\n", hs_strerror(err));
    }

    // 清理资源
    hs_free_scratch(scratch);
    hs_free_database(database);

    return 0;
}
```

四、扫描模式 Scanning

Hyperscan提供三种不同的扫描模式，每种模式都有自己的扫描功能，以`hs_scan`开头。此外，流模式具有用于管理流状态的多个其他API函数。

4.1 Handling Matches

所有这些函数都会在找到匹配项时调用用户提供的回调函数。此函数具有以下签名：即上面3.9中示例process_match：

static int process_match(unsigned int id, unsigned long long from, unsigned long long to, void *ctx);

```
typedef int (*match_event_handler)(unsigned int id, unsigned long long from,
unsigned long long to, unsigned int flags, void *context);
typedef int(*match_event_handler)(unsigned int id, unsigned long long
from,unsigned long <span id=52> longto, unsigned int flags,void*context);

    1、id参数将被设置为编译时提供的匹配表达式的标识符
    2、to参数将被设置为匹配的结束偏移量。
    3、如果为模式请求了SOM（请参见匹配开始），则from参数将设置为匹配的最左边可能的开始偏移。
    4、匹配回调函数能够通过（返回非零值）来停止扫描。// 常用 #define HS_SCAN_TERMINATED
   (-3)
```

4.2 Streaming Mode

Hyperscan流式传输运行时API的核心由打开、扫描和关闭Hyperscan数据流的函数组成：

- `hs_open_stream()`：分配并重新分配新的流以进行扫描。
- `hs_scan_stream()`：扫描给定流中的数据块，在检测到匹配项时引发匹配项。
- `hs_close_stream()`：完成给定流的扫描（如果流的末尾还有匹配未报告，它们将会在调用`hs_close_stream()`时被处理）并释放流状态。调用`hs_close_stream`后，流句柄无效，不应再次用于任何地方。

扫描数据时在数据中检测到的任何匹配都将通过函数指针回调返回给调用应用程序。

match-callback-function 能够通过返回一个**非零值来停止对当前数据流的扫描**。在流模式下，这样做的结果是流将处于无法扫描更多数据的状态，并且该流的任何后续`hs_scan_stream（）`调用都将立即返回`HS_SCAN_TERMINATED`。调用方仍然必须调用`hs_close_stream（）`来完成该流的清理过程。

流存在于Hyperscan库中，因此模式匹配状态可以跨多个目标数据块来进行维护，如果不维护此状态，则无法检测跨这些数据块的模式。然而，**这确实是以每个流需要一定量的存储为代价的（这个存储的大小在编译时是固定的）**，并且在某些情况下管理状态会带来轻微的性能损失。

虽然Hyperscan始终支持多个匹配的严格排序，但流匹配不会在当前流写入之前的偏移处交付，但零宽度断言除外，其中`\B`和`$`等结构可能会导致流写入的最后一个字符上的匹配延迟到下一个流写入或流关闭操作（因为可能该单词刚好在block末尾被截断）。

4.2.1 Stream Management

除了`hs_open_stream（）`、`hs_scan_stream（）`和`hs_close_stream（）之外`，Hyperscan API还提供了许多用于流管理的其他函数：

- `hs_reset_stream()`：将流重置为其初始状态;这等效于调用`hs_close_stream（）`，但不会释放用于流状态的内存。
- `hs_copy_stream()`：构造流的（新分配的）副本。
- `hs_reset_and_copy_stream()`：将一个流的副本构造到另一个流中，首先重置目标流。这个调用避免了由`hs_copy_stream（）`完成的分配。

4.2.2 Stream Compression

详细解释

1. **流对象的内存分配**：
    - 在 Hyperscan 中，流对象（stream object）是分配为一个固定大小的内存区域。这个大小是经过精心计算的，以确保在扫描操作期间不需要额外的内存分配。这样做的目的是提高扫描操作的效率，避免在运行时进行动态内存分配，从而提高性能。
2. **内存压力和压缩**：
    - 当系统处于内存压力较大的状态时，可能需要减少内存的使用量，尤其是对于那些预计不会很快使用的流对象。
    - 为了优化内存使用，Hyperscan 提供了将流对象转换为压缩表示的功能。压缩表示的流对象相比于完整的流对象，占用更少的内存，因为它不保留那些在当前流状态下不需要的组件。
3. **压缩表示**：
    - 压缩表示的流对象与完整的流对象不同，因为它仅包含当前流状态所需的部分，不包括额外的内存预留。这样可以节省内存，特别是在流对象不活跃或未被频繁使用时。
4. **API 函数**：

- Hyperscan 提供了一些 API 函数，用于将流对象转换为压缩表示，以及将压缩表示转换回完整的流对象。这些函数的具体实现可以帮助在内存紧张的情况下管理流对象的内存使用。

相关 API 函数

虽然具体的 API 函数没有在你提供的文本中列出，但常见的 Hyperscan 函数用于流对象压缩和解压通常包括：

- `hs_compress_stream()`：
  - 用于将当前的流对象转换为压缩表示。
- `hs_decompress_stream()`：
  - 用于将压缩表示的流对象转换回完整的流对象，以便继续使用。
- **hs_expand_stream():**
  - 此函数将压缩表示的流解压缩为新的流对象（新内存），可以直接用于扫描等操作。
- **hs_reset_and_expand_stream()**
  - 用于在不需要额外内存分配的情况下，将现有的流对象重置为新的流对象，适用于需要在内存有限的情况下优化流对象的重置和恢复操作。

总结

- **流对象**：在 Hyperscan 中，流对象是一个固定大小的内存区域，用于高效的流式匹配操作。
- **内存优化**：在内存压力较大时，可以通过将流对象转换为压缩表示来节省内存。压缩表示只保留当前状态所需的部分，减少了不必要的内存占用。
- **API 功能**：Hyperscan 提供了函数来实现流对象的压缩和解压，以帮助管理内存资源。

```
注意：
## 出于性能原因，不推荐在每次扫描调用之间使用流压缩，因为在压缩表示和标准流之间转换需要时间。
```

4.3 Block Mode

块模式运行时API由一个函数组成：`hs_scan()`。这个函数使用编译后的模式识别目标数据中的匹配项，并使用函数指针回调与应用程序进行通信 (3.9的process_match)。

这个单一的`hs_scan()`函数本质上等同于调用`hs_open_stream()`，对`hs_scan_stream()`只进行单次调用，然后是`hs_close_stream()`，由此块模式操作不会引起所有与流相关的开销。

```
适用于一次性处理较小的、"完全的数据块的场景"。
当数据可以一次性加载到内存中时，块模式非常高效，因为它避免了流管理的开销。
```

4.4 Vectored Mode

矢量模式运行时API与块模式API一样，由一个函数`hs_scan_vector()`组成。此函数接受数据指针和长度的数组，便于按顺序扫描内存中不连续的一组数据块。

从调用者的角度来看，此模式将产生相同的匹配，就像数据块集（a）通过一系列流模式扫描，再顺序扫描数据块集（b）并将（a-b）复制到单个内存块中，然后以块模式扫描一样。

```
矢量模式避免了将多个数据块合并到一个大的数据块中，这样可以节省内存和避免不必要的内存拷贝。
对于需要处理"大量分散数据的场景"，矢量模式可以更有效地利用内存和提高处理效率
```

4.5 Scratch Space 暂存空间

扫描数据时，Hyperscan需要少量临时内存来存储动态内部数据。不幸的是，这个数量太大而不能放在堆栈上，特别是对于嵌入式应用程序，并且动态分配内存太昂贵，因此必须为扫描功能提供预分配的“暂存”空间。

函数`hs_alloc_scratch()`分配足够大的暂存空间区域以支持给定的数据库。如果应用程序使用多个数据库，则只需要一个暂存区域：在这种情况下，对每个数据库（使用相同的`暂存`指针）调用`hs_alloc_scratch()`将确保暂存空间足够大，可以支持对任何给定数据库进行扫描。

虽然Hyperscan库是可反复调用的，但暂存空间的使用不是。例如，如果在设计上认为有必要运行递归或嵌套扫描（例如：match回调函数（3.9）），则需要为该上下文提供额外的暂存空间。

在没有递归扫描的情况下，每个线程需要一个这样的空间，并且可以（实际上应该）在开始数据扫描之前分配。

在一组表达式由一个“主”线程编译，数据将由多个“工作”线程扫描的情况下，可以使用函数`hs_clone_scratch()`允许为每个线程创建现有暂存空间的多个副本（而不是强制调用者多次通过`hs_alloc_scratch()`传递所有编译的数据库）。

```
hs_error_t err;
hs_scratch_t *scratch_prototype = NULL;
err = hs_alloc_scratch(db, &scratch_prototype);
if (err != HS_SUCCESS) {
    printf("hs_alloc_scratch failed!");
    exit(1);
}

hs_scratch_t *scratch_thread1 = NULL;
hs_scratch_t *scratch_thread2 = NULL;

err = hs_clone_scratch(scratch_prototype, &scratch_thread1);
if (err != HS_SUCCESS) {
    printf("hs_clone_scratch failed!");
    exit(1);
}
err = hs_clone_scratch(scratch_prototype, &scratch_thread2);
if (err != HS_SUCCESS) {
    printf("hs_clone_scratch failed!");
    exit(1);
}

hs_free_scratch(scratch_prototype);

/* Now two threads can both scan against database db,
   each with its own scratch space. */
```

4.6 Custom Allocators 自定义内存分配器

默认情况下，Hyperscan在运行时使用的结构（暂存空间、流状态等）是使用默认的系统分配器分配的，通常是`malloc()`和`free()`。

Hyperscan API提供了一种更改此行为的工具，以支持使用自定义内存分配器的应用程序。

These functions are: 这些职能是：

- `hs_set_database_allocator()`，它设置用于编译模式数据库的分配和释放函数。
- `hs_set_scratch_allocator()` 它设置用于暂存空间的分配和释放函数。
- `hs_set_stream_allocator()`，它设置用于流模式中的流状态的分配和释放函数。
- `hs_set_misc_allocator()`，它设置用于杂项数据（如编译错误结构和信息字符串）的分配和释放函数。

函数`hs_set_allocator()`可用于将所有自定义分配器设置为相同的分配/释放对。

> 所以，我们甚至可以使用dpdk的内存分配器来加速hyperscan。

五、Serialization 序列化

对于某些应用程序，在使用前立即编译Hyperscan模式数据库并不是一种适当的设计。有些用户可能希望：

- 在不同的主机上编译模式数据库;
- 将编译的数据库持久化到存储中，并且仅在模式改变时重新编译模式数据库;
- 控制已编译数据库所在的内存区域。

Hyperscan 模式数据库 在内存中并不是完全平坦的：它们包含指针，并有特定的对齐要求。**因此，它们不能直接复制（或以其他方式重新定位）**。为了支持这些用例，Hyperscan提供了序列化和非序列化编译模式数据库的功能

API提供以下功能：

1. `hs_serialize_database()`：将模式数据库序列化为可重定位的字节缓冲区。
2. `hs_deserialize_database()`：根据`hs_serialize_database()`的输出重新构造新分配的模式数据库。
3. `hs_deserialize_database_at()`：根据`hs_serialize_database()`的输出在给定的内存位置重建模式数据库。
4. `hs_serialized_database_size()`：给定一个序列化模式数据库，返回数据库在序列化时所需的内存块大小。
5. `hs_serialized_database_info()`：给定序列化模式数据库，返回包含数据库信息的字符串。此调用类似于`hs_database_info()`。

```
Hyperscan 在 deserialization 时的相关处理需要检查执行版本和平台兼容性。

hs_deserialize_database() 和 hs_deserialize_database_at() 函数：
    1、与 serialize 操作是相同版本的Hyperscan；
    2、数据库必须是使用当前主机平台支持的功能编译的。# 例如不同平台：AVX2 指令集支持度是不一样的
仅对满足上述两种情况的数据进行数据库化。
```

5.1 序列化步骤

开发环境中的序列化和压缩

```
// 伪代码示例
hs_serialize_database(database, &buffer, &buffer_size);
hs_compress_database(buffer, buffer_size, &compressed_buffer, &compressed_size);
```

生产环境中的解压缩和反序列化

```
// 伪代码示例
hs_expand_database(compressed_buffer, compressed_size, &expanded_buffer,
&expanded_size);
hs_deserialize_database(expanded_buffer, expanded_size, &database);
```

5.2 示例

5.2.1 开发环境端

这个端包含模式的编译、序列化和压缩。

```
#include <hs/hs.h>
#include <stdio.h>
#include <stdlib.h>

void serialize_and_compress_database(hs_database_t *database) {
    // 序列化数据库
    size_t serialized_size;
    void *serialized_data;
    hs_serialize_database(database, &serialized_data, &serialized_size);

    // 压缩序列化的数据
    size_t compressed_size;
    void *compressed_data;
    hs_compress_database(serialized_data, serialized_size, &compressed_data,
&compressed_size);

    // 将压缩后的数据写入文件
    FILE *file = fopen("database_compressed.bin", "wb");
    fwrite(compressed_data, 1, compressed_size, file);
    fclose(file);

    // 释放内存
    free(serialized_data);
    free(compressed_data);
}

int main() {
    // 定义模式和相应的 ID
    const char *patterns[] = {"foo", "bar"};
    unsigned int ids[] = {1, 2};
    unsigned int flags[] = {HS_FLAG_CASELESS, HS_FLAG_CASELESS};

    // 编译模式
```

```
    hs_database_t *database;
    hs_compile_error_t *compile_err;
    hs_error_t err = hs_compile_multi(
        2, patterns, ids, flags,
        HS_MODE_BLOCK, NULL, &database, &compile_err
    );

    if (err != HS_SUCCESS) {
        fprintf(stderr, "Failed to compile patterns: %s\n", compile_err-
>message);
        hs_free_compile_error(compile_err);
        return 1;
    }

    // 进行序列化和压缩操作
    serialize_and_compress_database(database);

    // 释放数据库
    hs_free_database(database);
    return 0;
}
```

5.2.2 生产环境端

这个端包含解压缩、反序列化和模式匹配。

```
cCopy code#include <hs/hs.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// 回调函数，用于处理匹配
static int match_handler(unsigned int id, unsigned long long from, unsigned long
long to, unsigned int flags, void *ctx) {
    printf("Match id: %u, from: %llu, to: %llu\n", id, from, to);
    return 0; // 继续扫描
}

void decompress_and_deserialize_database() {
    // 从文件读取压缩的数据库数据
    FILE *file = fopen("database_compressed.bin", "rb");
    if (!file) {
        fprintf(stderr, "Failed to open compressed database file.\n");
        return;
    }

    fseek(file, 0, SEEK_END);
    size_t compressed_size = ftell(file);
    fseek(file, 0, SEEK_SET);

    void *compressed_data = malloc(compressed_size);
    if (fread(compressed_data, 1, compressed_size, file) != compressed_size) {
        fprintf(stderr, "Failed to read compressed database from file.\n");
        fclose(file);
        free(compressed_data);
```

```
        return;
    }
    fclose(file);

    // 解压缩数据库数据
    size_t expanded_size;
    void *expanded_data;
    hs_expand_database(compressed_data, compressed_size, &expanded_data,
&expanded_size);
    if (expanded_data == NULL) {
        fprintf(stderr, "Failed to decompress database.\n");
        free(compressed_data);
        return;
    }
    free(compressed_data);

    // 反序列化数据库
    hs_database_t *database;
    hs_deserialize_database(expanded_data, expanded_size, &database);
    if (database == NULL) {
        fprintf(stderr, "Failed to deserialize database.\n");
        free(expanded_data);
        return;
    }

    // 释放内存
    free(expanded_data);

    // 进行模式匹配
    hs_scratch_t *scratch;
    hs_error_t err = hs_alloc_scratch(database, &scratch);
    if (err != HS_SUCCESS) {
        fprintf(stderr, "Failed to allocate scratch space.\n");
        hs_free_database(database);
        return;
    }

    const char *data = "foo bar baz";
    size_t data_len = strlen(data);

    err = hs_scan(database, data, data_len, 0, scratch, match_handler, NULL);
    if (err != HS_SUCCESS) {
        fprintf(stderr, "Failed to scan data.\n");
    }

    hs_free_scratch(scratch);
    hs_free_database(database);
}

int main() {
    decompress_and_deserialize_database();
    return 0;
}
```

5.3 libhs

Hyperscan 提供了两种不同的库版本来支持不同的使用场景：

5.3.1. 主 Hyperscan 库 (libhs)

- **包含内容**：这个库包含了编译器和运行时部分。
- **编译器部分**：用于将正则表达式模式编译成 Hyperscan 数据库。编译器部分是用 C++ 编写的，因此需要 C++ 语言链接，并依赖于 C++ 标准库。
- **运行时部分**：用于在应用程序中执行模式匹配。这部分功能支持扫描操作，也包含在主库中。

5.3.2. 运行时专用库 (libhs_runtime)

- **包含内容**：这个库仅包含运行时部分，即用于执行模式匹配的功能。
- **依赖性**：不依赖于 C++ 标准库，因此在嵌入式系统等环境中可以使用，而不需要 C++ 语言的支持。
- **使用场景**：适用于那些只需要执行模式匹配操作而不需要编译模式的应用场景。编译工作通常在其他主机上完成，然后将序列化后的模式数据库传递到运行时环境中。

使用场景

- **开发环境**：
  - **主 Hyperscan 库 (libhs)**：在开发环境中，开发人员会使用这个库来编译正则表达式模式。它可以处理编译和运行时操作，适合开发和测试阶段。
- **生产环境**：
  - **运行时专用库 (libhs_runtime)**：在生产环境中，尤其是在嵌入式系统中，可能不希望或不能使用 C++ 标准库。在这种情况下，使用 `libhs_runtime` 可以减少对系统资源的要求，并且只需要处理模式匹配而不需要编译。

总结

- **主库 (libhs)**：适用于需要编译和匹配功能的完整环境。
- **运行时库 (libhs_runtime)**：适用于仅需要模式匹配功能的环境，特别是嵌入式系统中。

> 许多嵌入式应用程序只需要Hyperscan库的扫描（“libhs_runtime”）部分。在这些情况下，模式编译通常在另一个主机上进行，然后将序列化的模式数据库交给应用程序以使用；

六、Performance Considerations 性能考量

Hyperscan在所有三种扫描模式（块、流、矢量）下都支持广泛的模式。它具有极高的性能水平，但某些模式可能会显着降低性能。

下面的指导原则将有助于构建性能更好的模式和模式集：

6.1 正则表达式构造

> 不要手动优化正则表达式构造。

大量的正则表达式可以用多种方式编写。例如，`/abc/i`的无大小写匹配可以写为：

- `/[Aa][Bb][Cc]/`
- `/(A|a)(B|b)(C|c)/`
- `/(?i)abc(?-i)/`

- `/abc/i`

Hyperscan能够处理所有这些结构。除非有特殊的原因，否则不要将模式从一种形式重写为另一种形式。此更改不会提高性能或减少管理费用。

6.2 Library usage

> 不要手动优化库的使用。

Hyperscan库能够处理小型写入、异常大小的模式集等。除非在使用该库时存在特定的性能问题，否则最好以简单直接的方式使用Hyperscan。例如，将库的输入缓冲到更大的块中不太可能有太大的好处，除非流式写入很小（比如，一次1-2个字节）。

Hyperscan还提供了高吞吐量匹配，每个核心一个控制线程;如果数据库在Hyperscan中以3.0 Gbps运行，这意味着3000位数据块将在1微秒内在一个控制线程中扫描。
对于要求在扫描22个3000位数据块，如果线程支持并行22线程，则也只需要一秒。因此，通常不需要缓冲数据以提供具有可用并行性的Hyperscan。

Hyperscan 和线程的高性能结合主要依赖于以下几个方面：

6.2.1 补充-并行化处理

1. 线程安全

Hyperscan 本身是线程安全的，意味着它可以在多个线程中同时使用。例如，你可以在不同线程中并行地执行匹配操作。为了在多线程环境中利用 Hyperscan，需遵循以下原则：

- **数据库共享**：`hs_database_t` 对象可以在多个线程中共享，因为它是线程安全的。但是，每个线程应该有自己的 `hs_scratch_t` 对象，用于存储扫描状态信息。
- **独立的扫描状态**：每个线程应分配并使用独立的 `hs_scratch_t` 对象。这是因为 `hs_scratch_t` 对象不是线程安全的，它存储了扫描状态和结果。

2. 并行化扫描

在多线程应用中，你可以并行执行 Hyperscan 的匹配操作，通过以下方式提升性能：

- **数据分片**：将待扫描的数据分成多个片段，每个线程处理一个数据片段。这种方法可以有效利用多核 CPU 的并行处理能力。
- **线程池**：使用线程池来管理线程，可以减少线程创建和销毁的开销，提升效率。

3. 线程与 Hyperscan 的并行匹配

下面是一个简单的示例，演示如何在多线程环境中使用 Hyperscan 执行并行扫描：

```
#include <hs/hs.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// 假设的匹配处理函数
static int match_handler(unsigned int id, unsigned long long from, unsigned long
long to, unsigned int flags, void *ctx) {
    printf("Match id: %u, from: %llu, to: %llu\n", id, from, to);
    return 0; // 继续扫描
```

```
}

// 每个线程的扫描任务
void *scan_thread(void *arg) {
    // 从参数中提取数据
    hs_database_t *database = ((hs_database_t **)arg)[0];
    const char *data = ((const char **)arg)[1];
    size_t data_len = ((size_t *)arg)[2];
    hs_scratch_t *scratch = ((hs_scratch_t **)arg)[3];

    // 执行扫描
    hs_error_t err = hs_scan(database, data, data_len, 0, scratch,
match_handler, NULL);
    if (err != HS_SUCCESS) {
        fprintf(stderr, "Thread scan failed: %d\n", err);
    }

    return NULL;
}

int main() {
    // 定义模式和 ID
    const char *patterns[] = {"foo", "bar"};
    unsigned int ids[] = {1, 2};
    unsigned int flags[] = {HS_FLAG_CASELESS, HS_FLAG_CASELESS};

    // 编译模式
    hs_database_t *database;
    hs_compile_error_t *compile_err;
    hs_error_t err = hs_compile_multi(
        2, patterns, ids, flags,
        HS_MODE_BLOCK, NULL, &database, &compile_err
    );
    if (err != HS_SUCCESS) {
        fprintf(stderr, "Failed to compile patterns: %s\n", compile_err-
>message);
        hs_free_compile_error(compile_err);
        return 1;
    }

    // 数据块
    const char *data_blocks[] = {
        "foo data segment",
        "bar data segment",
    };
    size_t data_lengths[] = {strlen(data_blocks[0]), strlen(data_blocks[1])};

    // 分配并初始化每个线程的 scratch 对象
    hs_scratch_t *scratch[2];
    for (int i = 0; i < 2; ++i) {
        err = hs_alloc_scratch(database, &scratch[i]);
        if (err != HS_SUCCESS) {
            fprintf(stderr, "Failed to allocate scratch space.\n");
            hs_free_database(database);
            return 1;
```

```
        }
    }

    // 创建线程
    pthread_t threads[2];
    void *args[4] = {database, NULL, NULL, NULL};

    for (int i = 0; i < 2; ++i) {
        args[1] = (void *)data_blocks[i];
        args[2] = (void *)&data_lengths[i];
        args[3] = scratch[i];
        pthread_create(&threads[i], NULL, scan_thread, args);
    }

    // 等待线程完成
    for (int i = 0; i < 2; ++i) {
        pthread_join(threads[i], NULL);
    }

    // 释放资源
    for (int i = 0; i < 2; ++i) {
        hs_free_scratch(scratch[i]);
    }
    hs_free_database(database);
    return 0;
}
```

4. 性能优化

- **批处理**：将多个数据块批量提交给 Hyperscan 进行扫描，可以减少多次调用的开销。
- **优化线程数**：根据硬件资源调整线程数量，避免线程过多导致的上下文切换开销。

5. 内存管理

- **有效的内存分配**：避免在多线程中频繁创建和销毁 `hs_scratch_t` 对象。可以使用线程池技术来复用 `hs_scratch_t` 对象。
- **内存一致性**：确保多线程操作中的数据一致性，避免竞争条件和数据冲突。

6.3 Block-based matching

> 在可能的情况下，优先选择基于块的匹配，而不是流匹配

每当输入数据出现在离散记录中，或者已经需要某种转换（例如URI规范化）**可以在处理之前积累所有数据时**，应该以块而不是流模式进行扫描。

不必要地使用流式模式会减少可以在Hyperscan中应用的优化数量，并可能使某些模式运行得更慢。

如果存在“块”和“流”模式的混合，则应在单独的数据库中扫描这些模式，除非流模式的数量远远超过块模式模式

6.4 Unnecessary databases

避免不必要的“联合”数据库（‘union’ databases）

如果有5种不同类型的网络流量T1到T5必须针对5个不同的规则签名集进行扫描，那么构建5个单独的数据库并针对适当的数据库扫描流量将比合并所有5个签名集并在事后删除不适当的匹配要有效得多。

即使在签名之间存在大量重叠的情况下也是如此。只有当签名的公共子集非常大时（例如，90%的签名出现在所有5种流量类型中），才应该考虑合并所有5个签名集的数据库，并且只有在出现在公共子集之外的特定模式没有强烈的性能要求时才考虑。

6.5 Allocate scratch ahead of time

在调用扫描函数之前，不要为模式数据库分配暂存空间。相反，应该在模式数据库编译或重新编译之后执行此操作。

Scratch分配并不一定是一种轻松的操作。由于这是第一次使用模式数据库（在编译或重新编译之后），Hyperscan在 `hs_alloc_scratch()` 中执行一些验证检查，并且还必须分配内存。

因此，确保在应用程序的扫描路径中，不在 `hs_scan()` 之前调用 `hs_alloc_scratch()` 非常重要（例如）。

相反，应该在模式数据库被编译或重新编译之后立即分配scratch，然后为以后的扫描操作保留。

甚至，你可以自己构建和维护自己的 scratch-pool。

6.6 Allocate one scratch space per scanning context

1、可以只分配一个暂存空间，以便它可以与许多数据库中的任何一个一起使用。(如下方示例，在一个线程中的多个数据库)
2、每个并发扫描操作（如多线程）都需要自己的暂存空间。 (6.2.1上的示例)

`hs_alloc_scratch()` 函数可以接受现有的暂存空间，并将其“增长”以支持使用另一个模式数据库进行扫描。这意味着不是为应用程序使用的每个数据库分配一个暂存空间，而是可以调用 `hs_alloc_scratch()`，并将指针指向相同的 `hs_scratch_t`，它将被适当地调整大小以用于任何给定的数据库。

你可以查看6.2.1上的示例。但那其实是针对一个db，需要创建了两个scratch空间；但对于多个数据库而言，我们可以只使用一个scratch：

```
hs_database_t *db1 = buildDatabaseOne();
hs_database_t *db2 = buildDatabaseTwo();
hs_database_t *db3 = buildDatabaseThree();

hs_error_t err;
hs_scratch_t *scratch = NULL;
err = hs_alloc_scratch(db1, &scratch); // scratch的地址都是一样的
if (err != HS_SUCCESS) {
    printf("hs_alloc_scratch failed!");
    exit(1);
}
```

```
err = hs_alloc_scratch(db2, &scratch); // scratch的地址都是一样的
if (err != HS_SUCCESS) {
    printf("hs_alloc_scratch failed!");
    exit(1);
}
err = hs_alloc_scratch(db3, &scratch); // scratch的地址都是一样的
if (err != HS_SUCCESS) {
    printf("hs_alloc_scratch failed!");
    exit(1);
}

/* scratch may now be used to scan against any of the databases db1, db2, db3.
    现在可以使用 scratch 扫描任何一个数据库 db1、db2 和 db3 。
*/

// 对第一个数据库进行匹配
err = hs_scan(db1, data, data_len, 0, scratch, match_handler, NULL);
if (err != HS_SUCCESS) {
    printf("Failed to scan data with db1\n");
}

// 对第二个数据库进行匹配
err = hs_scan(db2, data, data_len, 0, scratch, match_handler, NULL);
if (err != HS_SUCCESS) {
  printf("Failed to scan data with db2\n");
}

// 对第三个数据库进行匹配
err = hs_scan(db3, data, data_len, 0, scratch, match_handler, NULL);
if (err != HS_SUCCESS) {
   printf("Failed to scan data with db3\n");
}

// 释放资源
hs_free_scratch(scratch);
```

6.7 Anchored patterns 锚定模式

（if a pattern is meant to appear at the start of data, be sure to anchor it）
如果一个模式是要出现在数据的开始、结尾，一定要使用锚定模式。

锚定模式（`/^.../`）的匹配比其他模式简单得多，特别是锚定到缓冲区（或流，在流模式下）开始的模式。将整个模式扩展到缓冲区的末尾，则会导致性能降低（因为大部分的锚点匹配其实都是有或多或少的长度限制的，适当的限制缓冲区匹配范围，可以提高性能），尤其是在流模式下。

有多种方法可以将模式锚到特定偏移：

- `^`和`\A`构造将模式锚到缓冲区的开头。例如，`/^foo/`*只能，也使用 **HS_FLAG_ANCHORED** 标志来限制模式仅从数据流的开头进行匹配。
- `$`、`\z`和`\Z`构造将模式锚到缓冲区的末尾。例如，`/foo\z/`只能在被扫描的数据缓冲区以`foo`结尾时匹配。(应该注意的是，`$`和`\Z`也会在缓冲区末尾的换行符之前匹配，所以`/foo\z/`会匹配`abcfoo`或`abcfoo\n`。

- `min_offset`和`max_offset`扩展参数还可以用于限制模所能匹配的位置。例如，`max_offset`为10的模式`/foo/`仅在缓冲区中小于或等于10的偏移处匹配。(这个模式也可以写成`/^.{0,7}foo/`，使用`HS_FLAG_DOTALL`标志编译）。

以下是关于锚定模式及其使用方法的整理表格：

锚定位置	描述	使用方法	示例模式
开头	限制模式只能从数据流的开头开始匹配	使用`HS_FLAG_ANCHORED`标志	`hs_compile("foo.*", HS_FLAG_ANCHORED, HS_MODE_BLOCK, NULL, &database, NULL);`
结尾	限制模式只能在数据流的结尾匹配	模式中直接包含`$`符号	`hs_compile(".*bar$", 0, HS_MODE_BLOCK, NULL, &database, NULL);`
中间位置	无直接的锚定支持，但可以通过设计模式或自定义逻辑模拟	模式中嵌入前置条件或使用分块扫描	(1)`hs_compile("prefix.*target", 0, HS_MODE_BLOCK, NULL, &database, NULL);` (2) **使用 max_offset 等相关偏移限制。具体可以查看3.5节**

6.8 Matching everywhere

```
Avoid patterns that match everywhere, and remember that our semantics are ‘match everywhere, end of match only’.
避免到处匹配的模式，match everywhere的语义是“程序从数据开始即进行匹配，直到数据结束”。
```

以下是对上述说明的举例和表格梳理：

说明	详细解释	示例模式	数据流	匹配结果
避免匹配所有位置的模式	避免使用会在数据流每个位置都尝试匹配的模式。这类模式可能导致性能问题，因为会在每个位置进行匹配。	`.*`	`"abcdefg"`	在每个位置尝试匹配，性能较差。
“匹配所有位置，仅报告匹配的结束位置”	Hyperscan 默认行为是报告匹配的结束位置，而不是开始位置。这意味着只报告匹配的结束点。	`foo.*bar`	`"foobarxyz"`	报告`"foobar"`匹配的结束位置。

例如： 输入数据`0123 abcddddd`将匹配`/abc/`一次，但匹配`/abcd*/`五次（在`abc`、`abcd`、`abcddd`、`abcddd`和`abcddddd`处）。

6.9 Bounded repeats in streaming mode 有界重复结构

> 有界重复在流模式中是代价高昂的。应避免大量和不必要地使用有界重复

有界重复结构，如`/x.{1000，1001}abcd/`在流模式下是非常昂贵的。它要求我们对每个`x`字符采取行动（相对于搜索更长的字符串来说，这本身就很昂贵），并可能记录`x`发生的数百个偏移量的历史，以防`x`和`abcd`字符被流边界分隔开。

应避免大量和不必要地使用有界重复，特别是在规则签名的其他部分非常特定的情况下，避免在正则表达式中过度使用有界重复可以简化模式，提高匹配效率。。例如，匹配病毒有效载荷的病毒签名可能是足够的，而无需预先包括前缀，该前缀包括例如2个字符的Windows可执行前缀和有界重复。

6.10 Prefer literals 首选文字

> 在正则表达式中，使用尽可能多的字符量（`literals`）要比使用更复杂的模式更有效率。
> 这一原则尤其适用于流式模式（`streaming mode`），其中较长的字符量更优于较短的字面量。

1. **字面量(字符量)的优势**：
   - **速度**：匹配包含字面量的正则表达式通常比匹配不包含字面量的正则表达式运行得更快。这是因为字面量模式在数据中很容易定位，相比之下，更复杂的模式可能需要额外的计算或比较。
   - **匹配效率**：较长的字面量（例如 14 个字符的字面量）可以更准确地定位数据，从而减少匹配的负担。较短的字面量（例如 4 个字符的字面量）可能会导致更多的匹配结果，从而增加处理的复杂性。
2. **流式模式的特别注意**：
   - 在流式模式下，模式的效率更加依赖于模式中的字面量。如果模式中较长的字面量出现在较早的位置，它能够更快地过滤数据流中的无关部分，从而提高整体性能。

示例 1：较长的字面量

正则表达式：

```
/wab\d*\w\w\w/
```

- **解释**：匹配以 `w`，`a`，`b` 开头，后面跟随任意数量的数字，然后是三个字母字符的字符串。

效率：

- 这个模式在匹配时需要较少的计算，因为它包含了几个字面量和一个字符集合（`\w`）。

示例 2：较短的字面量

正则表达式：

```
/w\w\d*\w\w/
```

- **解释**：匹配以字母字符开头，后面跟随一个字母字符、任意数量的数字，最后是两个字母字符的字符串。

效率：

- 这个模式虽然也能匹配特定模式，但由于没有包含长字面量，它可能会导致更多的匹配尝试，从而性能较差。

示例 3：隐式字面量

正则表达式：

```
/[0-2][3-5].*\w\w/
```

- **解释**：匹配 0 到 2 和 3 到 5 之间的数字组合，后面跟随任意字符和两个字母字符。

效率：

- 这个模式有效地包含了九个 2 字符的字面量，即使这些字面量是隐式的，也能提高匹配效率。

示例 4：模式中的长字面量

正则表达式：

```
/blah\w*foobar/
```

- **解释**：匹配包含 `blah` 和 `foobar` 字符串的模式，中间可以有任意数量的字母字符。

效率：

- 由于 `blah` 和 `foobar` 都是长字面量，这个模式在流式模式下的性能优于只包含较短字面量的模式，例如 `/b\w*foobar/`。

示例 5：模式中的短字面量

正则表达式：

```
/b\w*fo/
```

- **解释**：匹配以 `b` 开头，后面跟随任意数量的字母字符，然后是 `fo` 的字符串。

效率：

- 这个模式包含短字面量，不如包含较长字面量的模式高效。

总结

- **优先使用字面量**：优先使用长字面量可以提高正则表达式的匹配速度。
- **流式模式中的长字面量**：在流式模式中，将较长的字面量放在模式的前面是更有效的策略。
- **避免复杂模式**：复杂的模式（尤其是不需要的有界重复）会导致性能下降，应尽量简化正则表达式。

6.11 “Dot all” mode

```
尽可能使用“全点”模式：HS_FLAG_DOTALL
```

“Dot All” 模式 使得正则表达式中的点号 (`.`) 匹配所有字符，包括换行符。在某些情况下，启用此模式可以显著提高性能，特别是在处理多行数据时。

- **作用**：
  - 在未启用 `HS_FLAG_DOTALL` 标志时，正则表达式中的 `.` 默认不匹配换行符。因此，像 `/A.*B/` 这样的模式会被解释为 `/A[^\n]*B/`，即 `A` 和 `B` 之间不能包含换行符。

    - 启用 `HS_FLAG_DOTALL` 标志后，`.*` 将匹配包括换行符在内的所有字符。这意味着 `/A.*B/` 可以匹配包含换行符的文本。
- **性能考虑**：
    - **启用 DOTALL**：启用 `HS_FLAG_DOTALL` 时，扫描 `/A.*B/` 将更直接，因为 `.` 可以匹配所有字符。这通常会减少正则表达式引擎的复杂性，提高扫描效率。
    - **不启用 DOTALL**：在未启用 DOTALL 的情况下，正则表达式引擎需要在每次匹配时处理换行符，从而可能会增加额外的开销。特别是当正则表达式中包含许多点号时（如 `.*`），处理换行符会导致性能下降。
- **使用场景**：
    - 如果模式需要匹配跨越多行的内容，**例如在日志文件中查找模式，启用 “dot all” 模式是有益的。**
    - **对于只处理单行数据的情况，不启用 `HS_FLAG_DOTALL` 可能更合适**，尤其是当数据块是按行处理时。

示例：假设你要匹配以下日志行中的某个模式：

```
Line1
Line2
MatchHere
Line4
```

你希望匹配从 "Line1" 开始到 "MatchHere" 的所有内容，包括换行符。

不使用 `HS_FLAG_DOTALL`：

正则表达式 `/Line1.*MatchHere/` 在没有 `HS_FLAG_DOTALL` 的情况下将被解释为 `/Line1[^\n]*MatchHere/`。它会尝试匹配 `Line1` 和 `MatchHere` 之间不包含换行符的内容，因此无法匹配跨行内容。

使用 `HS_FLAG_DOTALL`：

正则表达式 `/Line1.*MatchHere/` 启用 `HS_FLAG_DOTALL` 后，它将匹配包括换行符在内的所有内容，从而可以成功匹配跨越多行的内容

6.12 Single-match flag 单次匹配

`HS_FLAG_SINGLEMATCH` 是 Hyperscan 提供的一个**单次匹配标志**，用于限制每个模式在扫描过程中最多只能匹配一次。这种限制可以带来一些性能优化，但也有可能在特定情况下影响性能。

解释

- **功能**：
    - 当启用 `HS_FLAG_SINGLEMATCH` 标志时，Hyperscan 会在数据流中仅报告每个模式的第一次匹配。这意味着一旦模式匹配成功，后续的数据将不会再次匹配该模式。
- **优化**：
    - **性能改进**：使用单次匹配标志可以减少匹配过程中所需的状态跟踪和计算，进而提高匹配效率。这在处理大型数据流或高吞吐量场景中尤其有效。
    - **状态空间减少**：单次匹配减少了需要存储的匹配状态，从而减少内存消耗和计算开销。
- **开销**：

- **跟踪开销**：启用此标志会增加一些开销，因为系统需要跟踪每个模式的匹配状态。如果应用程序中的匹配频率较低，可能会因为额外的状态跟踪而导致性能下降。

使用场景

- **高吞吐量场景**：在数据流中，如果每个模式只需要匹配一次，例如检测是否存在某种特定模式或签名，使用 `HS_FLAG_SINGLEMATCH` 可以有效提高处理速度。
- **资源受限场景**：在内存或计算资源有限的环境中，使用单次匹配标志可以减少内存消耗和计算负担。

示例

假设我们有一个模式集合用于检测网络数据流中的特定模式，我们希望在每个数据块中只报告一次每个模式的匹配结果。可以使用 `HS_FLAG_SINGLEMATCH` 来实现这一点。

编译模式时启用单次匹配标志的示例代码：

```
#include <hs.h>

// 编译模式
hs_database_t *compile_patterns() {
    hs_database_t *db = NULL;
    hs_compile_error_t *compile_err;
    hs_scratch_t *scratch = NULL;
    const char *patterns[] = {"pattern1", "pattern2"};
    hs_platform_info platform = {0};
    hs_expr_ext_t ext = {0};

    // 设置扩展参数
    ext.flags = HS_EXPR_FLAG_SINGLEMATCH; // 启用单次匹配标志

    // 编译模式
    hs_compile_multi(patterns, NULL, NULL, 2, HS_MODE_STREAM, &platform, &ext,
&compile_err, &db);

    if (compile_err) {
        printf("Compile error: %s\n", compile_err->message);
        hs_free_compile_error(compile_err);
        return NULL;
    }

    return db;
}

// 执行扫描
void scan_data(hs_database_t *db, const char *data, size_t length) {
    hs_scratch_t *scratch = NULL;
    hs_error_t err;
    hs_iterate_t iterate = NULL;

    // 分配内存
    hs_alloc_scratch(db, &scratch);

    // 扫描数据
    err = hs_scan(db, data, length, 0, scratch, iterate, NULL);
    if (err != HS_SUCCESS) {
```

```
        printf("Scan error: %d\n", err);
    }

    // 释放资源
    hs_free_scratch(scratch);
}
```

注意：在此示例中，我们通过 `hs_compile_multi` 函数编译模式，并使用 `HS_FLAG_SINGLEMATCH` 来启用单次匹配标志。这样，当扫描数据时，每个模式在整个数据流中只会报告一次匹配结果。

6.13 Start of Match (SOM) 标志

Start of Match (SOM) 标志 是一个用于请求匹配的开始位置的标志。在某些情况下，获取匹配的起始位置可能会很昂贵，尤其是在流模式下，因为它需要大量的状态来跟踪流中的匹配位置。

- **作用**：
  - 当启用 `HS_FLAG_SOM_LEFTMOST` 时，Hyperscan 会报告匹配的最左侧起始位置。这意味着除了返回匹配的结束位置外，Hyperscan 还会返回匹配的起始位置。
  - 如果模式需要知道匹配的起始位置（例如，处理特定格式的日志或数据），则可以启用这个标志。
- **性能考虑**：
  - **性能开销**：请求 SOM 信息可能会增加性能开销，并需要更多的流状态来存储在流模式下的匹配信息。这是因为需要额外的计算和存储来追踪每个匹配的起始位置。
  - **替代方案**：在某些情况下，使用有界重复（例如 `/foo.{300}bar/`）或者在应用程序中后处理匹配长度可能比启用 SOM 标志更轻量级。
- **示例**：
  - 如果你只需要知道匹配的结束位置，而不需要起始位置，可以避免使用 `HS_FLAG_SOM_LEFTMOST`，这样可以减少性能开销。
  - 对于长模式，例如 `/foo.*bar/`，如果使用 SOM 标志会显著增加开销，尤其是在流模式下，因为需要检查整个匹配的起始位置。此时，使用较短的匹配模式（例如有界重复 `/foo.{300}bar/`）或通过其他方法检查匹配长度可能更高效。

6.14 Approximate matching 近似匹配

近似匹配是一个实验性功能。
- 如果可以通过精确匹配满足需求，且性能是主要关注点，则应避免使用近似匹配。
- 在性能关键的应用场景中，如实时数据流处理或高吞吐量的匹配任务，使用近似匹配可能会带来不必要的性能开销。

由于匹配的特异性降低，通常会产生与近似匹配相关的性能影响。这种影响可能会因为匹配模式、编辑距离的不同而有很大差异。

- **近似匹配**：
  - 近似匹配允许模式匹配那些与目标数据有一定差异的字符串。比如，它可以处理带有错别字的文本，或者对模式进行轻微的变异。
  - 近似匹配通常通过指定**编辑距离**或**汉明距离**来实现，编辑距离是指将一个字符串转变为另一个字符串所需的最少操作数（插入、删除、替换），而汉明距离则是对两个等长字符串之间不同

字符的计数。

- **性能影响**：
  - 由于近似匹配增加了模式的模糊性，相比于精确匹配，性能可能会受到影响。尤其是在处理复杂模式或较大编辑距离时，匹配的计算量增加可能导致较大的性能开销。
  - 具体的性能影响会根据模式的复杂性和设置的编辑距离不同而有所变化。

是否应避免使用近似匹配

- **使用情境**：
  - 如果你的应用场景需要处理可能存在变异或错别字的字符串，近似匹配是一个有用的工具。例如，在搜索引擎中查找拼写错误的关键词，或者在数据清洗过程中处理可能的格式变异。
  - 对于那些对性能要求严格的场景，或者匹配精度非常高的场景，可能需要谨慎使用近似匹配。
- **避免使用**：
  - 如果可以通过精确匹配满足需求，且性能是主要关注点，则应避免使用近似匹配。
  - 在性能关键的应用场景中，如实时数据流处理或高吞吐量的匹配任务，使用近似匹配可能会带来不必要的性能开销。

七、工具

7.1 hscheck

`hscheck` 工具允许用户快速检查 Hyperscan 是否支持一组模式。如果模式被 Hyperscan 的编译器拒绝，则会在标准输出中提供编译错误。

7.2 hsbench

hsbench 是 Hyperscan 提供的一个性能基准测试工具，主要用于测量和评估 Hyperscan 数据库在不同配置下的性能。这对于了解 Hyperscan 在实际使用中的表现，以及在特定硬件和数据条件下的性能是非常有用的。

主要功能

1. **基准测试**：`hsbench` 用于测量 Hyperscan 数据库的匹配性能，包括扫描速度、匹配率等。可以帮助确定模式和数据库在不同负载下的表现。
2. **性能分析**：生成详细的性能数据，包括匹配延迟、吞吐量等指标。可以用于优化模式和配置，以提高性能。
3. **配置灵活性**：支持多种配置选项，以便在不同的测试环境和条件下运行基准测试。

工具输出结果

- **性能指标**：`hsbench` 将输出包括吞吐量、延迟、匹配率等性能指标。
- **报告文件**：结果将保存到指定的报告文件中，可以用于进一步分析和优化。

7.3 hscollider

hscollider 是一个工具，主要用于生成、测试和调试 Hyperscan 数据库的合成测试数据。它是一个命令行工具，通常用于帮助开发者创建有效的模式和验证数据库的行为。

此外，**hscollider** 提供了两个 CMake 目标来运行更大规模的扫描测试，利用 Hyperscan 自带的测试模式进行验证。以下是如何使用这些 CMake 目标来执行测试的说明：

Make Target （make执行命令）	Description
`make collide_quick_test`	Tests all patterns in streaming mode. 流模式
`make collide_quick_test_block`	Tests all patterns in block mode. 块模式

7.4 hsdump

当构建在调试模式下（使用 CMake 指令`CMAKE_BUILD_TYPE`设置为 `Debug`）时，Hyperscan 支持在使用 `hsdump` 工具进行模式编译期间转储有关其内部的信息。

显示数据库内容: 输出数据库中的模式、模式 ID、以及模式的其他属性。
调试和验证: 帮助开发者检查数据库是否按预期工作，或者诊断编译或匹配过程中可能出现的问题。
数据库信息: 显示有关数据库结构的信息，如模式的组合方式、标志设置等。

八、Chimera

Chimera 是一种混合正则表达式匹配引擎，结合了 Hyperscan 和 PCRE 的特性。以下是 Chimera 的详细说明：

8.1 Chimera 的设计目标

1. **全面支持 PCRE 语法**:
    - Chimera 旨在支持 PCRE（Perl Compatible Regular Expressions）语法的所有功能。这包括支持 PCRE 特有的正则表达式特性，如回溯、复杂的模式匹配、分组、非捕获组等。
2. **利用 Hyperscan 的高性能**:
    - Hyperscan 是一个高性能的正则表达式匹配引擎，以其高速、并行处理能力著称。Chimera 结合了 Hyperscan 的性能优势，提供了高效的正则表达式匹配。

Chimera 的主要特性

- **混合引擎**:
    - Chimera 将 PCRE 的灵活性和 Hyperscan 的性能结合在一起，使其能够处理复杂的正则表达式模式，同时保持高效的匹配速度。
- **性能优化**:
    - 通过利用 Hyperscan 的优化技术，Chimera 能够处理大规模数据集和复杂的模式匹配，同时保持较高的处理速度。
- **兼容性**:
    - 支持 PCRE 的全面语法意味着用户可以使用熟悉的正则表达式模式，而不必担心与 Hyperscan 的性能特性不兼容。

使用 Chimera 的场景

- **复杂模式匹配**:
    - 当需要支持复杂的正则表达式模式（如嵌套的分组、回溯等）并且要求高性能时，Chimera 是一个理想的选择。
- **高性能应用**:

- 在需要处理大量数据或高频率匹配的应用中，Chimera 的高性能特性可以显著提高处理效率。

8.2 Chimera 与 Hyperscan 性能比较

Chimera 和 Hyperscan 都是高性能的正则表达式匹配引擎，但它们在设计目标和性能特点上有所不同。以下是两者的主要性能比较：

特性	Hyperscan	Chimera
设计目标	高性能的模式匹配引擎，专注于处理大规模数据和并行计算。	结合 Hyperscan 的性能优势和 PCRE 的语法支持。
语法支持	支持 Hyperscan 的特有语法，基本不支持 PCRE 的复杂语法。	支持完整的 PCRE 语法，包括回溯、分组、复杂模式等。
性能	设计上强调极高的扫描速度，尤其是在大数据量的并行处理上。	性能高于单纯的 PCRE 实现，但可能稍逊于纯 Hyperscan 实现，取决于具体的模式。
匹配方式	使用固定模式的并行匹配，优化了吞吐量和延迟。	综合了 Hyperscan 的速度和 PCRE 的灵活性，适用于复杂模式的高性能匹配。
资源使用	高效使用 CPU 和内存资源，特别是在大数据量和高并发场景中表现出色。	可能需要更多的资源来支持 PCRE 复杂的语法，但仍然具有较高的性能表现。
应用场景	适用于大规模数据处理、高吞吐量需求的场景，如网络流量分析、安全监控等。	适用于需要复杂正则表达式支持的场景，如日志分析、数据提取等。
流式处理	支持高效的流式处理，适用于实时数据分析和处理。	支持流式处理，但可能在处理非常复杂的模式时比纯 Hyperscan 实现稍慢。

性能比较详细说明

1. **吞吐量和延迟**:
   - **Hyperscan**: 在设计上高度优化，特别是在大规模数据流和并发处理场景下表现优异。由于其专注于快速的模式匹配和并行处理，Hyperscan 能够在高吞吐量场景下实现低延迟。
   - **Chimera**: 在性能上接近 Hyperscan，但由于其支持 PCRE 复杂语法，可能在处理某些模式时稍慢。例如，PCRE 的回溯机制可能会影响性能，尤其是在复杂的正则表达式模式下。
2. **语法支持**:

- **Hyperscan**: 主要支持其自有的正则表达式语法，不支持 PCRE 的高级特性。适合需要高性能匹配但不需要复杂正则功能的应用。
- **Chimera**: 支持 PCRE 的完整语法，包括回溯、非捕获组等复杂特性，使其在需要高度灵活性的场景中更具优势。

3. **资源使用**:
   - **Hyperscan**: 优化了 CPU 和内存使用，特别是在处理大量数据时。其设计理念是最大化资源的利用效率。
   - **Chimera**: 由于支持更复杂的正则表达式，可能需要更多的资源来处理这些模式。尽管如此，它仍然在性能和资源使用之间保持了良好的平衡。
4. **应用场景**:
   - **Hyperscan**: 更适合需要处理大量数据流、实时监控或高吞吐量的场景，如网络安全、流量分析等。
   - **Chimera**: 更适合需要复杂模式匹配的场景，如日志分析、文本数据提取等。

总结

- **Hyperscan** 在性能上通常优于 Chimera，特别是在处理高吞吐量和大规模数据流的场景中。它的专有语法和优化使其在这些场景下表现出色。
- **Chimera** 提供了 PCRE 的完整语法支持，使其能够处理更复杂的模式，但可能在性能上略逊于纯 Hyperscan 实现。它适用于需要复杂匹配的应用场景。