**facebook开源库folly介绍**

[QQ客户端团队博客](http://ju.outofmemory.cn/feed/132/) 2012-08-27 **1932** 阅读

[C/C++](http://ju.outofmemory.cn/tag/C%2FC%2B%2B/)

　　2012年6月，全球最大的社交网站facebook向外界开源了其后台使用的基础组件库folly，folly即为英文 Facebook Open-source LibrarY 的缩写。folly是一个提供类似 std 和 boost 功能的底层库，并且基于最新的c++ 11特性，folly的开发者包括了 Andrei Alexandrescu之类的c++编程领域的大师级人物，想必folly的含金量不会低。既然folly提供了类似std和boost这种基础库的功能，那为何不直接使用std和boost呢?用官方的话说，folly是为了实用性和效率而设计的，在大规模的场景中具有良好地表现，是folly的统一主题。而且facebook也承认为了追求效率，会导致有些设计会比较特别。换句话说就是，folly与std和boost的最大不同就是在大规模地场景里效率更高。

**folly的逻辑结构**

　　从逻辑结构上看folly的设计，folly由一些相对比较独立的组件构成，如字符串组件:string.h、FBString.h；哈希：hash.h、AtomicHashMap.h、AtomicHashArray.h； 类似varint的动态数据类型:dynamic.h等等。可以看出这些组件彼此之间比较独立，但是folly并没有对这些组件之间的依赖关系做任何限制，也就是说folly中的组件可以互相引用，相互依赖。

**folly的组织结构**

　　folly的目录结构采用了类似boost等库使用的“结巴”设计，即”folly/folly/”下面才是源码,目录结构如下图所示:

folly目录结构

* **detail** 包含了一些组件的实现细节和辅助类型，内部使用，外部不应该依赖它
* **docs**一些组件的简单介绍和使用示例
* **experimental**facebook内部使用的，带有试验性质的代码，不稳定
* **test** 一些测试代码

*folly/folly目录下平坦地存放着供外部使用的组件源码，总体目录结构类似其命名空间结构。*

**folly的兼容性**

　　由于使用了c++11的一些特性，所以必须要gcc 4.6以及以上的编译器，并且由于使用了gcc编译器相关的一些代码，如gcc内建函数\_\_builtin\_expect，gcc内联汇编实现的cas等等,所以编译器也只得用gcc咯；并且虽然某些组件针对32bit有作处理如fbstring，但是很多组件只能在x64下使用，如PackedSyncPtr。所以环境需要一个64位的linux,gcc4.6+即可。

**主要的组件概览**

***Malloc.h***

　　Malloc.h顾名思义是实现内存分配相关代码的文件，在Malloc.h中主要的“特别设计”是在Realloc内存块的时候，没有直接使用realloc,而是首先判断当前环境是否有jemalloc,据说jemalloc在多线程环境下的内存分配效率要比默认的内存分配器性能更好，jemalloc的realloc函数是rallocm。folly在realloc内存的时候首先用rallocm尝试以ALLOCM\_NO\_MOVE的形式realloc，如果失败，就直接使用malloc然后memcopy。

*上图中的usingJEMalloc使用了弱符号判断当前环境下是否有rallocm(jemalloc的特殊api)这个函数符号，以此来判断是否可以使用jemalloc。*

在代码中判断rallocm!=NULL即可。

上面的代码是当能够使用jemalloc时的样子，如果不能使用jemallo怎么办呢？逻辑其实还是差不多，都是判断是否有可能在realloc的时候move内存，如果需要，则使用malloc+memcpy代替，如果不需要，直接realloc。当不能使用rallocm时，folly通过判断当前内存使用量是否超过了内存分配量(相对于需要relloc的内存块),如果超过一半，则认为realloc有可能会move内存，否则直接调用realloc。

***SmallLock.h***

　　folly既然是用于高并发环境，那么就自然离不开同步类型了，这个头文件实现了一个sleeper,MicroSpinLock以及PicoSpinLock三个类型。

* **Sleeper**: 既然是锁那么就需要wait了,Sleeper其实啥事也没做，就是在等锁的时候循环4000次，如果还没有获得锁，那就sleep 0.5ms吧。
* **MicroSpinLock**: 顾名思义MicroSpinLock就是很小的自旋锁…..小表现在什么地方呢。它只使用了一个字节中的最低一位来存储锁的状态，即FREE=0,LOCKED=1。实现起来也很简单，我们来看看其cas操作实现(Compare And Swap):
* cas操作是用来查询锁状态－符合条件－改变锁状态的原子操作，MicroSpinLock使用了GCC内联汇编来实现了CAS。我们来分析下上面的代码，从asm volatile开始看起，asm不用多说了表示内嵌汇编，volatile告诉编译器不要优化咱的汇编代码。再来看括号里面的部分。括号里面被3个冒号(“:”)分割成了4部分：

|  |  |
| --- | --- |
| 第一部分(lock开始) | 这一段字符串表示具体的汇编指令，指令之间以分号隔开 |
| 第二部分(“=r”(out)) | 这一段表示第一部分的指令的输出，”=”号表示out为只写，”r”表示使用通用寄存器,同时 out对应汇编指令中的%0,因为是第一个参数 |
| 第三部分(“a”(compare)开始) | 这一段是输入，即将compare放到al寄存器,将newVal放到一个通用寄存器中，而r存放的是lock\_的地址，分别对应%1,%2,%3 |
| 第四部分(“memory”开始) | “memory”表示汇编指令修改了内存，”flags”表示标志寄存器被修改了 |

* 那么明白了内联汇编的大概语法后，我们就明白了这段代码的含义：这段代码拿 comapre变量的值同lock\_变量比较是否相等，如果相等则把newval的值赋给lock\_,然后zf标志会被置位，并通过setz设置结果out。即当我们想获得锁的时候这样用：cas(0,1); 返回true表示成功。
* **PicoSpinLock:**
* PicoSpinLock也是一个自旋锁，与MicroSpinLock最大的不同是其可以指定存储锁的数据大小，支持2,4,8字节三种大小.锁的标志位位于数据的最高位,也就是说，这个PicoSpinLock中除锁占用的最高位外，其他bit位可以用来存储数据，这种锁在PackedSyncPtr中有特殊用处。因为对锁数据的存储和MicroSpinLock有所区别，PicoSpinLock使用了另外一种方式实现CAS:
* 和MicroSpinLock的CAS实现大同小异，只不过是cmpxchgb换成了bts(#size是根据情况接上w,l,q等不同长度后缀的),bts指令的功能是将％2中的第％1位取出放到CF里面，并将该位置1;setnc功能是根据cf的值设置返回结果。即ret==1则表示置为之前锁的状态是free，反之是locked。释放锁的时候使用了btr指令，其他都差不多。

***PackedSyncPtr.h***

　　PackedSyncPtr从功能上讲应该叫做同步指针，它使用了前文中提到的PicoSpinLock<uintptr\_t>包装一个64位的指针，PackedSyncPtr只支持x86-64架构的处理器，原因嘛，先看看PackedSyncPtr的结构:

　　看上图可以知道，PackedSyncPtr一共有8字节的数据，其中最高位用作自旋锁的标志，低48位用作保存指针的值，而其余的用于存储自定义的数据。相信大家有疑问，为什么可以石无忌惮地占用一个64位指针的高16位？而不会产生其他问题？前面说了，PackedSyncPtr只支持x86-64架构，我们来看看x86-64的wiki怎么说：

|  |
| --- |
| *尽管虚拟地址有64位的宽度，但当前的实现机制（以及任何已知处在计划阶段的芯片）并不允许整个16EB的虚拟地址空间都被使用。大多数的操作系统和应用程序在可见的未来都用不到如此巨大的地址空间（比如，Windows在AMD64上的实现仅应用了16TB，即44位的宽度），所以实现如此巨大的地址宽度只会增加系统的复杂度和地址转换的成本，带不来任何好处。AMD因此决定，在对这一架构的首次实现中，只有虚拟地址的最低48位才会在地址转换（页表查询）时被使用。* |

<http://zh.wikipedia.org/wiki/X86-64>

　　也就是说x86-64架构的实现者AMD定下了规则－只使用64位指针的低48位，所以我们可以大胆地占用其余16位了。

***RWSpinLock.h***

　　RWSpinLock实现了读写自旋锁，folly的读写锁具有3种锁状态：READER(读)，WRITER(写) 以及 UPGRADED(更新)。

1. **Reader:**表示读锁，同时可以允许多个实例获取读锁，并排斥Writer；
2. **Writer:**表示写锁，只有当没有Reader和Upgraded锁时，才能获取到写锁，并且当写锁被持有时，排斥其他实例获取3种锁；
3. **Upgraded:**表示更新锁，用在这样一种场景下：读取数据，根据数据的值决定是否修改数据。Upgraded排斥其他upgraded以及writer锁，但是其他实例可以获取Reader锁。当没有upgraded锁的时候，如果你想读取a，并根据a的值决定是否修改a，那么你只能在整个过程中保持writer锁，此时其他实例只有等锁了。而如果有upgraded锁，你可以在读取a时保持upgraded锁，然后读取a，注意，此时其他实例可以获取reader锁，当你需要修改a时，做一个原子操作释放upgraded锁&获取writer锁，即只有在真正需要写的时候才获取writer锁。

上面讲了RWSpinLock的使用原理和状态转换，下面来看看关键代码的实现,首先是锁标志的存储，RWSpinLock使用一个int32\_t来存储所有的锁状态，各种锁状态的定义如下:

READER可以累加，UPGRADED和WRITER同时只有一个被持有，所以可以看成是int32\_t的最低两位用来存放UPGRADED和WRITER锁，高30位用来累加READER(由于每次都是按4累加，不会影响低两位)

下面看下lock函数的实现：

try\_lock后面分析，这里的大致逻辑是循环try\_lock，当尝试次数大于1000次后，释放当前线程的调度，让操作系统去调度其他线程。

我们来看看LIKELY宏是什么：

可以看出LIKELY和UNLIKELY其实是使用了gcc的一个内建函数\_\_builtin\_expect,该宏的作用是优化if语句的跳转指令，LIKELY表示if语句里面的代码执行的概率更高，编译器就会对生成的跳转代码进行优化，使程序在该处大部分情况下不需要跳转，从而提高了cache命中率，也就提高了效率。

再看看try\_lock的实现：

cas使用了 c++ 11里的 std::atomic<T>::compare\_exchange\_strong函数，这段代码的意思只有当前不存在任何锁被持有的时候获取writer锁。第三个参数 std::memory\_order\_acq\_rel是atomic提供的内存序约束功能。

try\_lock\_shared就是Reader锁，我们看看Reader锁的获取方式：

reader锁的获取方法是不管锁的状态如何，先累加4(即增加一个读锁)，如果发现已经有writer锁存在，那就把4再减回去。

其他加锁解锁的操作都差不多，读者可以自己看看RWSpinLock.h的源码。

***FBString.h***

　　fbstring.h重新实现了string的核心－fbstring\_core，fbstring\_core实现了和字符串内存操作有关的大部分算法，内存分配则采用了Malloc.h实现的几个优化过的分配函数。fbstring\_core的实现与我们以往对字符串的认识有所不同，首先根据字符串长度，folly将字符串分为小，中，大三种(small,medium,large), 其特点是：

1. **small类型**字符串要求长度是23字节以内（包括终结符），其存储位置和fbstring\_core对象一致，看后面的代码就明白了。
2. **medium类型**字符串要求长度在24-254,字符串数据存储在堆上，也就是说fbstring\_core持有其数据的指针，而不是直接存储字符串
3. **large类型**字符串要求长度>=255字节,存储方式和medium类似

看了上面的代码可以确定，small直接存储在fbstring\_core对象中。medium和large类型的字符串长度存放在size\_变量中，capacity\_的高两位存放了字符串的种类(small,medium,large)，通过掩码读写，下面是相关常量和掩码的定义：

从两个extractmask的定义可以看出，capacity\_的最高两位用于存放字符串的类型，从Category中的定义可以看出，small=00,medium=10,large=01, 这样定义small=0有一个好处，那就是small字符串的长度直接等于capacity\_t的最高字节的值(其实存放的应该是capacity-size):

**拷贝策略:**

　　由于三种字符串的内存分配策略并不一致，在字符串拷贝时也会采用不同的策略。并且在字符串拷贝的时候也要考虑到并发和性能。

　　Small是直接存储在对象内部的，拷贝这样的字符串时是直接拷贝 MediumLarge结构,速度快,并且直接存储在对象内部时无需考虑多个对象共享内存的问题，更无需进行锁操作。而medium的拷贝过程则是: malloc & copy,即medium虽然是对象持有指针，但每次拷贝都是重新分配新内存，每个medium字符串对象之间也不会共享内存，和small一样，medium也不会有并发的问题。

　　再来看看large类型，large类型通常存储较大的字符串，如果每次拷贝都重新分配新的内存，在效率上肯定是不行的，所以large采用写时拷贝策略，而普通的拷贝仅仅是增加一次引用计数而已。即large类型的字符串对象之间是可以共享内存的，只要不对字符串进行修改，效率也是非常高的。

　　对于small和large都是很容易理解的，但是对于medium的存在就比较疑惑了，为什么不对medium也使用写时拷贝呢？其实在高并发，并行的计算下，medium类型的拷贝策略避免了同步的代价，相对于同步的开销，每次拷贝一个不算太大的内存还算比较快的。

　　fbstring\_core还实现了同字符串相关的一些算法如：预留内存算法reserve,收缩字符串shrink等等，处理的方式同上面拷贝构造函数大同小异，都是针对3种类型分别处理，同时在变更内存大小之后会有字符串类型的变化。如small升级为medium,large等情况。

　　分析了部分的folly实现之后，我们可以看到folly在实现上有着许多特别地设计，这些都说为了追求效率，而且是告并发与并行环境下的计算性能，这次的分析就到这里，剩下的组件接下来再继续分享。