Objective-C的Runtime机制是Mac和iOS程序中的核心，而objc_msgSend函数是Runtime的核心，进言之，这个函数的核心正是方法缓存。今天将代领大家探索苹果是如何以一种线程安全且不影响程序性能的方式来调整和分配方法缓存所用内存的，其所用的技术也许是在其他关于线程安全的资料中从未使用的。

消息转发的概念

Objc_msgSend方法的工作方式是为发送过来的方法查找恰当的方法实现，并且跳转到该实现中的方式工作。用官方的说法来讲，查找方法的过程是这样的：

```
IMP lookUp(id obj, SEL selector) {
    Class c = object_getClass(obj);
    while(c) {
        for(int i = 0; i < c->numMethods; i++) {
            Method m = c->methods[i];
            if(m.selector == selector) {
                return m.imp;
            }
        }
        c = c->superclass;
    }
    return _objc_msgForward;
}
```

【注】代码中的一些变量名已经替换，如果你对原始代码有兴趣可以去下载一份Objective-C runtime的[源码](http://www.opensource.apple.com/source/objc4/)

方法缓存

在Objective-C的程序中，消息发送随处可见，如果对每一条消息都执行完整的消息搜索，那将会使程序变得异常迟钝。

解决方法就是缓存，每一个类都拥有一个哈希表与之关联,这个哈希表将选择器映射到方法实现。使用哈希表的初衷正是为了最大限度的提高读取速度，同时 objc_msgSend 使用了极其细致且高效的汇编源码来快速执行哈希表的检索，这使得在缓存模式下的消息发送仅维持在一个个位数的纳秒量级上。当然，任何消息在第一次使用的时候很慢，以后将会非常快。

我们提到的缓存，是用来提高多次读取最近使用资源的速度的，它通常也是有大小限制的。例如，你可能会缓存从网络上加载的图片，这样连续2次读取图片就不会请求网络2次了。但是，你又不想使用太多内存，所以你可能会给缓存图片的数量设置一个最大值，当达到最大值，且又有新的图片进来时，就可以把最旧的图片删掉。

在大多数情况下这是一种很好的解决方案，但是在一些隐蔽的情况下可能会有较差的表现。举个例子，如果你把图片缓存个数设置为40个，但是应用却以41张一组这样的规模循环图片的话，你会忽然发现你的缓存策略失效了。

对于我们自己的应用，我们可以测试和调整缓存的大小以避免这种情况发生，但是Objective-C的Runtime机制并没有这种条件。因为方法缓存对于性能极其严苛，并且每个条目都相对较小。runtime并不会强制的限制缓存区的大小，相反，它会在需要的时候扩充缓存区以保存所有已经被发送的消息。

请注意，缓存有时候会刷新；有一些操作会造成缓存数据过期，如在处理过程中加载入更多的代码，或者改变一个类的方法列表，恰当的缓存区会被销毁并且允许再次填充的。

改变缓存大小、分配内存以及线程问题

在概念上，改变缓存的大小简单，就像这样

```
    bucket_t *newCache = malloc(newSize);
    copyEntries(newCache, class->cache);
    free(class->cache);
    class->cache = newCache;
```

Objective-C runtime 实际上在这里采用了一些捷径，但是不会将旧的条目拷贝到新的缓存区！毕竟它仅仅是缓存而已，没有必要保存数据，这些条目将在消息发送的时候再次被填充，所以，真实情况是这样的：

```
free(class->cache);
class->cache = malloc(newSize);
```

在单线程环境下，你需要做的仅有这些，那么这篇文章也本该很短。当然Objective - C runtime也必须要支持多线程，也就是说所有这些代码都必须是线程安全的。任何给出的类的缓存，都可以从多个线程中被同时访问，所以这些代码必须考虑周全，确保可以应付这种场景。

写到这里的代码是无法处理多线程的情况的。在`释放旧的缓存到分配新的缓存之前`这段时间内，其他线程也许会访问这些已经失效的缓存指针，这会造成它使用的数据是垃圾数据，或者由于指定的内存并未映像物理地址出而立刻崩溃。

我们该如何解决这种问题？典型的保存共享数据的一种方法是加锁，代码如下：

```
lock(class->lock);
free(class->cache);
class->cache = malloc(newSize);
unlock(class->lock);
```

为此，包括读取在内的所有访问都会被这个锁控制。也就是说 *Objc_msgSend* 方法需要获得这个锁，查找缓存，然后放锁。每次进行加锁，解锁操作都会增加许多开销，考虑到缓存每次对自己的检索只需要几纳秒时间，这对性能的影响太大了。

我们也许会尝试通过一些其他方式关闭这个时间窗口（*释放旧缓存到分配新缓存这个时间窗*）。例如，对缓存先分配地址并赋值，然后再去释放旧的缓存如何？

```
bucket_t *oldCache = class->cache;
class->cache = malloc(newSize);
free(oldCache);
```

这会有一些帮助，但是并不能解决这个问题。另外一个线程也许会检索旧的缓存指针，然后在他可以访问内容之前通过系统先行占取这块缓存。这块旧的缓存在其他的线程再次运行之前被销毁，之前的问题再次出现。

如果加一个像这样的延迟呢？

```
bucket_t *oldCache = class->cache;
```

```
class->cache = malloc(newSize);
after(5 /* seconds */, ^{
    free(oldCache);
});
```

这几乎是可行的。但还是有下面的情况，一个线程刚好被系统抢占了缓存，并且被抢占的时间足够长，这样延迟5秒的释放就会先触发。这使得崩溃的可能微乎其微,但也不能完全保证不会发生。

不采用一个随机的延迟时间，一直等待到时间窗完全腾出来会怎么样呢？我们对Objc_msgSend加一个计数器：

```
    gInMsgSend++;
    lookUpCache(class->cache);
    gInMsgSend--;
```

一个恰当的线程安全版本需要用到计数器的原子性，合适的内存`阻隔`来确保依赖加载/存储显示正常。本文的目的不是讨论这些，想象它们已经存在就好了。

在计数器的帮助下，缓存的再分配像是这样：

```
    bucket_t *oldCache = class->cache;
    class->cache = malloc(newSize);
    while(gInMsgSend)
        ; // spin
    free(oldCache);
```

注意到这里没有必要阻塞执行objc_msgSend方法就可以正常工作。一旦释放缓存的代码确定在它替换了缓存指针之后，objc_msgSend中没有东西了，这段代码就会继续向下执行，释放旧的缓存区。其他线程可能会在旧的缓存区指针释放的时候调用 Objc_msgSend 方法，但是这个相对较新的调用将不能再使用旧的指针，因此这种条件下是线程安全的。

不断的循环是低效率且不够优美的。释放缓存并没有那么紧急。释放内存是好的，如果要花些时间也没什么问题。与其低效的循环，不如让我们保存一份未释放缓存列表，每次当缓存释放的时候会将所有等待中的操作全部执行完毕，上代码:

```
bucket_t *oldCache = class->cache;
class->cache = malloc(newSize);
append(gOldCachesList, oldCache);
if(!gInMsgSend) {
    for(cache in gOldCachesList) {
       free(cache);
    }
    gOldCachesList.clear();
}
```

当一个新的发送消息在处理的过程中，这个操作不会立刻释放旧的缓存，但这并不是问题。当再次访问它、访问之后的时候、或者将来的某个时间点会被释放。

这个版本已经相当接近Objective-C Runtime机制的实际运行原理了。`

零耗费标志

这两个交互的部分存在这极大的不对称。Objc_msgSend这边, 可能每秒会运行百万次，并且的确是需要尽可能地快。最好的情况是单次调用的运行时间只需要几纳秒。另一方面，改变缓存区的大小是一个较少的操作，并且随着

app的持续运行将会变得越来越少。一旦应用达到了一种稳态，不在加载新的代码，或者编辑消息列表，并且缓存变得足够大而且能满足所需的时候，缓存块大小的重新计算操作将不会再发生。但在此之前，这个操作在缓存区增大到它所需的大小时或许会发生个几百或者几千次，但是与Objc_msgSend相比而言是极其小的，并且性能敏感性也更低。

由于这种不对称性，在消息发送方应该放尽可能少的任务，即使这会使缓存释放部分会变慢一些。在objc_msgSend的百万级别CPU循环中每削减一个CPU运行循环累积下带来的优势与释放操作是一个以巨大优势的净赢。

即使全局计数器花费太大。在objc_msgSend方法中的这两个附加的内存访问操作将仍然带来很大的开销。它们需要保持原子性并且使用内存隔离会使情况更糟。

幸运的是，Objective-C runtime机制有一个技术是以牺牲缓存释放的速度来将objc_msgSend的开销降为0。

假设全局计数器的目的在于追踪任何在一个特定代码区块内的线程。这些线程已经有已有一些来监测当前它们是在哪段代码中执行，它就是程序计数器（program counter）。这是一个CPU内部的寄存器，其功能在于记录当前指令的内存地址。与全局计数器相比，我们可以检查每个线程的程序计数器来确认他是否在执行objc_msgSend

。如果所有线程都没有执行objc_msgSend方法，那么对它而言，释放缓存就是安全的，代码实现如下:

```
    BOOL ThreadsInMsgSend(void) {
        for(thread in GetAllThreads()) {
            uintptr_t pc = thread.GetPC();
            if(pc >= objc_msgSend_startAddress && pc cache;
    class->cache = malloc(newSize);
    append(gOldCachesList, oldCache);
    if(!ThreadsInMsgSend()) {
        for(cache in gOldCachesList) {
            free(cache);
        }
        gOldCachesList.clear();
    }
```

然后，objc_msgSend不必做任何其他的事情。它可以直接访问缓存区，而不用给读取加个标志，就像下面这样:

```
lookUpCache(class->cache);
```

由于缓存释放需要检查进程中的每个线程的状态，因此它是相对低效的。但是如果objc_msgSend只用考虑单线程的环境下，它的执行效率将会非常高。这值得做出权衡。这基本上就是苹果的Runtime机制如何工作的。

实际的代码

到底苹果如何实现上述的技术可以在runtime的实现文件[objc-cache.mm]文件中的函数 _collection_in_critical 中找到。

关键的PC位置存储在全局变量中:

```
OBJC_EXPORT uintptr_t objc_entryPoints[];
OBJC_EXPORT uintptr_t objc_exitPoints[];
```

实际上objc_msgSend有多种实现（比如返回结构体版本的），并且内部的cache_getImp 函数也会直接访问缓存。这些都需要被检查，以确保释放缓存的安全性。

函数本身不需要参数，返回值是 **int**类型的，使用起来就像一个标志位一样，用来标识在一个关键函数中是否有多个线程：

```
1       static int _collectiong_in_critical(void)
2       {
```

为了专注于更好的代码，我将会略过这个函数中一些无聊的代码。如果你想看全部的代码，在[这里](http://www.opensource.apple.com/source/objc4/objc4-646/runtime/objc-cache.mm)可以找到。

获得线程信息的API位于mach层面。task_threads 获得了给定任务中所有线程的线程列表，并且这些代码使用它来获得其所在进程中的其他线程。

```
1       ret = task_threads(mach_task_self(), &threads, &number);
```

它返回了一组包含了多个thread_t值的threads数组，并且可以获得数组元素的个数，然后它会遍历这些元素

```
1       for (count = 0; count < number; count++)
2       {
```

取得一个线程的PC的操作在另外一个独立的函数中，我们可以简单看下：

```
1       pc = _get_pc_for_thread (threads[count]);
```

然后遍历这些入口和出口，然后比较各个元素

```
1               for (region = 0; objc_entryPoints[region] != 0; region++)
2               {
3                   if ((pc >= objc_entryPoints[region]) &&
4                       (pc <= objc_exitPoints[region]))
5                   {
6                       result = TRUE;
7                       goto done;
8                   }
9               }
10          }
```

在循环结束后向调用者返回结果

```
1           return result;
2       }
```

_get_pc_for_thread这个函数如工作？这是相对简单代码，它通过调用thread_get_state方法来获得目标线程的寄存器状态。它位于一个独立的函数中的主要原因是寄存器状态的结构是特定于系统架构的，因为每个架构下有着不同的寄存器。这就意味着这个函数对于每种支持的架构需要一个独立的实现，尽管每种实现都几乎是一样的。这里有一个关于x86-64架构下的实现

```
1       static uintptr_t _get_pc_for_thread(thread_t thread)
2       {
3           x86_thread_state64_t            state;
4           unsigned int count = x86_THREAD_STATE64_COUNT;
5           kern_return_t okay = thread_get_state (thread, x86_THREAD_STATE64, (thread_state
6           return (okay == KERN_SUCCESS) ? state.__rip : PC_SENTINEL;
```

```
    }
```

注意到`rip`是PC在x86-64架构下的名字，其中R代表"register",IP代表"instruction pointer";

入口点和出口点他们本身是在一个汇编语言文件中定义的，这个文件中同时还包含了问题中的一些其他函数，

```
    .private_extern _objc_entryPoints
    _objc_entryPoints:
        .quad   _cache_getImp
        .quad   _objc_msgSend
        .quad   _objc_msgSend_fpret
        .quad   _objc_msgSend_fp2ret
        .quad   _objc_msgSend_stret
        .quad   _objc_msgSendSuper
        .quad   _objc_msgSendSuper_stret
        .quad   _objc_msgSendSuper2
        .quad   _objc_msgSendSuper2_stret
        .quad   0
    .private_extern _objc_exitPoints
    _objc_exitPoints:
        .quad   LExit_cache_getImp
        .quad   LExit_objc_msgSend
        .quad   LExit_objc_msgSend_fpret
        .quad   LExit_objc_msgSend_fp2ret
        .quad   LExit_objc_msgSend_stret
        .quad   LExit_objc_msgSendSuper
        .quad   LExit_objc_msgSendSuper_stret
        .quad   LExit_objc_msgSendSuper2
        .quad   LExit_objc_msgSendSuper2_stret
        .quad   0
```

_collecting_in_critical 与我们上面假设的例子中的用法相似。它在释放残留的内存垃圾之前调用。runtime实际上有两种独立的模式：一种是留下垃圾知道下次再有其他线程进入临界函数。另一个是不断的循环直到清除干净，而且通常会同时释放这些垃圾内存。

```
    // Synchronize collection with objc_msgSend and other cache readers
    if (!collectALot) {
        if (_collecting_in_critical ()) {
            // objc_msgSend (or other cache reader) is currently looking in
            // the cache and might still be using some garbage.
            if (PrintCaches) {
                _objc_inform ("CACHES: not collecting; "
                              "objc_msgSend in progress");
            }
            return;
        }
    }
    else {
        // No excuses.
        while (_collecting_in_critical())
            ;
    }
    // free garbage here
```

第一种留下垃圾的模式是用于普通的缓存区重新计算的。通常会释放垃圾的循环的模式用于runtime的清除所有类的所有缓存,这很显然会产生喝多垃圾。通过对代码的分析，这仅会在打印所有调试信息的调试设备这种情况下才会发生。它会清除缓存，正是于消息缓存会干涉日志输出。

结论

性能和线程安全是一个矛盾体。不同的代码快访问共享数据要求更高的线程安全性这也是不平衡的。一个全局的标志或者计数器是一种利用这种特点的一种方法。在Objective-C的runtime机制中，苹果采用了比这种策略更深层次的方法，它通过使用每个线程的程序计数器（PC）隐式的表明了什么时候一个线程正在执行一种不安全的操作。这是一个特例，并且其他地方很难看到这种方法的用武之地，但它本身很奇妙。