

**xcgg** Lv1

2020年03月10日 阅读 436

[关注](#)

## iOS-锁-@synchronized

`@synchronized`，同步锁，又名对象锁，由于其使用简单，基本上是在 `iOS` 开发中使用最频繁的锁。

使用方式如下：

```
@synchronized() {  
    // 需要加锁的代码块  
}
```

OC

### 原理

那么 `@synchronized` 到底是如何实现了锁的功能呢？我们看一个例子：

```
- (void)synchronizedTest {  
    @synchronized (self) {  
        NSLog(@"====synchronized====");  
    }  
}
```

OC

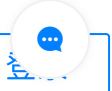
对程序设置一个断点，进入汇编，我们可以看到，发生变化的前后包裹了两个方法：

[首页](#) ▾[探索掘金](#)

```

objc_sync_enter
16 -> 0x10bba7b67 <+55>: movq    -0x8(%rbp), %rdi
17 0x10bba7b67 <+55>: movq    0x3a5a(%rip), %rsi      ; "ticketCount"
18 0x10bba7b6e <+62>: movq    0x24a3(%rip), %rcx      ; (void *)0x00007fff513f7780:
objc_msgSend
19 0x10bba7b75 <+69>: movl    %eax, -0x34(%rbp)
20 0x10bba7b78 <+72>: callq   *%rcx
21 0x10bba7b7a <+74>: movq    %rax, -0x40(%rbp)
22 0x10bba7b7e <+78>: jmp     0x10bba7b83              ; <+83> at ViewController.m
23 0x10bba7b83 <+83>: movq    -0x40(%rbp), %rax
24 0x10bba7b87 <+87>: cmpq    $0x0, %rax
25 0x10bba7b8b <+91>: jbe     0x10bba7c3e              ; <+270> at
ViewController.m:64:13
26 0x10bba7b91 <+97>: movq    -0x8(%rbp), %rax
27 0x10bba7b95 <+101>: movq    0x3a2c(%rip), %rsi      ; "ticketCount"
28 0x10bba7b9c <+108>: movq    0x2475(%rip), %rcx      ; (void *)0x00007fff513f7780:
objc_msgSend
29 0x10bba7ba3 <+115>: movq    %rax, %rdi
30 0x10bba7ba6 <+118>: movq    %rax, -0x48(%rbp)
31 0x10bba7baa <+122>: callq   *%rcx
32 0x10bba7bac <+124>: movq    %rax, -0x50(%rbp)
33 0x10bba7bb0 <+128>: jmp     0x10bba7bb5              ; <+133> at ViewController.m
34 0x10bba7bb5 <+133>: movq    -0x50(%rbp), %rax
35 0x10bba7bb9 <+137>: decq    %rax
36 0x10bba7bbc <+140>: movq    0x39ed(%rip), %rsi      ; "setTicketCount:"
37 0x10bba7bc3 <+147>: movq    0x244e(%rip), %rcx      ; (void *)0x00007fff513f7780:
objc_msgSend
38 0x10bba7bca <+154>: movq    -0x48(%rbp), %rdi
39 0x10bba7bce <+158>: movq    %rax, %rdx
40 0x10bba7bd1 <+161>: callq   *%rcx
41 0x10bba7bd3 <+163>: jmp     0x10bba7bd8              ; <+168> at ViewController.m
42 0x10bba7bd8 <+168>: xorl    %edi, %edi
43 0x10bba7bda <+170>: callq   0x10bba833e              ; symbol stub for: sleep
44 0x10bba7bdf <+175>: movl    %eax, -0x54(%rbp)
45 0x10bba7be2 <+178>: jmp     0x10bba7be7              ; <+183> at
ViewController.m:62:50
46 0x10bba7be7 <+183>: movq    -0x8(%rbp), %rdi
47 0x10bba7beb <+187>: movq    0x39d6(%rip), %rsi      ; "ticketCount"
48 0x10bba7bf2 <+194>: movq    0x241f(%rip), %rax      ; (void *)0x00007fff513f7780:
objc_msgSend
49 0x10bba7bf9 <+201>: callq   *%rax
50 0x10bba7bfb <+203>: movq    %rax, -0x60(%rbp)
51 0x10bba7bff <+207>: jmp     0x10bba7c04              ; <+212> at
ViewController.m:62:13
52 0x10bba7c04 <+212>: leaq    0x24dd(%rip), %rdi      ; @
53 0x10bba7c0b <+219>: xorl    %eax, %eax
54 0x10bba7c0d <+221>: movb    %al, %cl
55 0x10bba7c0f <+223>: movq    -0x60(%rbp), %rsi
56 0x10bba7c13 <+227>: movb    %cl, %al
57 0x10bba7c15 <+229>: callq   0x10bba82e4              ; symbol stub for: NSLog
58 0x10bba7c1a <+234>: jmp     0x10bba7c1f              ; <+239> at
ViewController.m:63:9
59 0x10bba7c1f <+239>: jmp     0x10bba7c5a              ; <+298> at ViewController.m
60 0x10bba7c24 <+244>: movl    %edx, %ecx
61 0x10bba7c26 <+246>: movq    %rax, -0x18(%rbp)
62 0x10bba7c2a <+250>: movl    %ecx, -0x1c(%rbp)
63 0x10bba7c2d <+253>: movq    -0x28(%rbp), %rdi
64 0x10bba7c31 <+257>: callq   0x10bba8338              ; symbol stub for:
objc_sync_exit
65 0x10bba7c36 <+262>: movl    %eax, -0x64(%rbp)

```


[首页](#)
[探索掘金](#)


```
objc_sync_enter
objc_sync_exit
```

或者使用下面代码对程序进行编译：

```
clang -x objective-c -rewrite-objc -isysroot /Applications/Xcode.app/Contents/Developer/Pla
```

```
appDelegateClassName = NSStringFromClass(((Class (*)(id, SEL))(void
*)objc_msgSend)((id)objc_getClass("AppDelegate"),
sel_registerName("class")));
{
    id _rethrow = 0;
    id _sync_obj = (id)appDelegateClassName;
    objc_sync_enter(_sync_obj);
    try {
        struct _SYNC_EXIT {
            _SYNC_EXIT(id arg) : sync_exit(arg) {}
            ~ _SYNC_EXIT() {objc_sync_exit(sync_exit);} id sync_exit;
        } _sync_exit(_sync_obj);

    } catch (id e) {_rethrow = e;}
}
{ struct _FIN { _FIN(id reth) : rethrow(reth) {}
  ~_FIN() { if (rethrow) objc_exception_throw(rethrow); }
  id rethrow;
  } _fin_force_rethrow(_rethrow);}
}
```

也可以得出，分析的重点应该是以下代码：

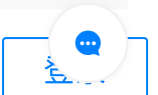
```
objc_sync_enter
objc_sync_exit
```

通过符号断点我们可以将上述代码定位到 `objc` 源码。

```
// Allocates recursive mutex associated with 'obj' if needed.
int objc_sync_enter(id obj)
{
    int result = OBJC_SYNC_SUCCESS;

    if (obj) {
        SyncData* data = id2data(obj, ACQUIRE);
        assert(data);
        data->mutex.lock();
    }
}
```

0C


[首页](#)
[探索掘金](#)


```

        _objc_inform("NIL SYNC DEBUG: @synchronized(nil); set a breakpoint on objc_sync
    }
    objc_sync_nil();
}

return result;
}

BREAKPOINT_FUNCTION(
    void objc_sync_nil(void)
);

```

从代码可以得出以下结论：

- `@synchronized` 使用的是递归锁 (recursive mutex)
- `@synchronized(nil)` 不会做任何事情，可以用来防止死递归。

我们再来看看当 `obj` 存在的时候，`@synchronized` 做了什么。

```
SyncData* data = id2data(obj, ACQUIRE);
```

OC

通过这行代码，可以看出来 `obj` 是以 `SyncData` 这种结构来保存的。`SyncData` 是一个结构体，具体信息如下：

```

typedef struct alignas(CacheLineSize) SyncData {
    struct SyncData* nextData;
    DisguisedPtr<objc_object> object;
    int32_t threadCount;
    recursive_mutex_t mutex;
} SyncData;

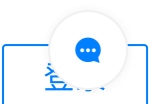
```

OC

- `struct SyncData* nextData` : `SyncData` 的指针节点，指向下一条数据
- `DisguisedPtr<objc_object> object` : 锁住的对象
- `int32_t threadCount` : 等待的线程数量
- `recursive_mutex_t mutex` : 使用的递归锁

获取 `SyncData` 结构的数据的流程是怎样的？

1. 如果支持 `tls` 缓存，从 `tls` 缓存获取 `obj` 的相关信息。该方法是检查每个线程单项快速缓存是否有匹配的对象。


[首页](#)
[探索掘金](#)


此处引入一个概念，`tls`，`Thread Local Storage`，线程局部存储，它是操作系统为线程单独提供的私有空间，通常只有有限的容量。

```
result = data;
lockCount = (uintptr_t)tls_get_direct(SYNC_COUNT_DIRECT_KEY);
lockCount++;
tls_set_direct(SYNC_COUNT_DIRECT_KEY, (void*)lockCount);
```

OC

此处如果多次进入，也就是递归操作，只会对 `lockCount` 进行加1操作。

如果获取到数据，说明对象又被加了一次锁，更新 `tls` 中存储的 `obj` 信息，锁的次数加1，并将数据返回。如果没有获取到，则进入第二步。

2. 在线程缓存 `SyncCache` 中查找是否存在 `obj` 的数据信息。该方法是检查已拥有锁的每个线程高速缓存中是否有匹配的对象。

```
typedef struct {
    SyncData *data;
    unsigned int lockCount; // number of times THIS THREAD locked this block
} SyncCacheItem;

typedef struct SyncCache {
    unsigned int allocated;
    unsigned int used;
    SyncCacheItem list[0];
} SyncCache;

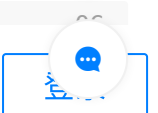
SyncCache *cache = fetch_cache(NO);
SyncCacheItem *item = &cache->list[i];
item->lockCount++;
```

OC

如果存在当前 `obj` 的数据信息，将线程缓存 `SyncCache` 中的 `obj` 的锁的次数加1，并将数据返回。如果没找到就进入第3步。

3. 在使用列表 `sDataLists` 中查找对象

在列表 `sDataLists` 中查找，需要对查找过程加锁，防止在多线程查找导致的异常。使用列表 `sDataLists` 把 `SyncData` 又做了一层封装，元素是一个结构体 `SyncList`。

[首页](#) ▼[探索掘金](#)

```
using spinlock_t = mutex_tt<LOCKDEBUG>;
#define LOCK_FOR_OBJ(obj) sDataLists[obj].lock
#define LIST_FOR_OBJ(obj) sDataLists[obj].data
struct SyncList {
    SyncData *data;
    spinlock_t lock;

    constexpr SyncList() : data(nil), lock(fork_unsafe_lock) { }
};
static StripedMap<SyncList> sDataLists;
```

遍历，进行匹配：

```
SyncData* p;
SyncData* firstUnused = NULL;
for (p = *listp; p != NULL; p = p->nextData) {
    if ( p->object == object ) {
        result = p;
        OSAtomicIncrement32Barrier(&result->threadCount);
        goto done;
    }
    if ( (firstUnused == NULL) && (p->threadCount == 0) )
        firstUnused = p;
}
```

如果找到，就将数据写入 **tls** 缓存和线程缓存 **SyncCache**，并返回数据。

```
// 写入tls缓存
tls_set_direct(SYNC_DATA_DIRECT_KEY, result);
tls_set_direct(SYNC_COUNT_DIRECT_KEY, (void*)1);

// 写入线程缓存
if (!cache) cache = fetch_cache(YES);
cache->list[cache->used].data = result;
cache->list[cache->used].lockCount = 1;
cache->used++;
```

OC

4. 创建一个新的 **SyncData** 放入 **sDataLists** 中，并存入 **tls** 缓存和线程缓存中，然后返回。

```
posix_memalign((void **)&result, alignof(SyncData), sizeof(SyncData));
result->object = (objc_object *)object;
```


[首页](#)
[探索掘金](#)


```
result->nextData = *listp;
*listp = result;
```

看完了获取锁，我们再来看看释放锁。释放的过程和保存相似。如果传入的对象是空的，也不会做任何事情。

```
// End synchronizing on 'obj'.
// Returns OBJC_SYNC_SUCCESS or OBJC_SYNC_NOT_OWNING_THREAD_ERROR
int objc_sync_exit(id obj)
{
    int result = OBJC_SYNC_SUCCESS;

    if (obj) {
        SyncData* data = id2data(obj, RELEASE);
        if (!data) {
            result = OBJC_SYNC_NOT_OWNING_THREAD_ERROR;
        } else {
            bool okay = data->mutex.tryUnlock();
            if (!okay) {
                result = OBJC_SYNC_NOT_OWNING_THREAD_ERROR;
            }
        }
    } else {
        // @synchronized(nil) does nothing
    }

    return result;
}
```

OC

如果传入的对象有值：

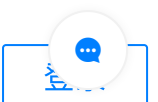
1. 先从 **tls** 缓存中查找，如果找到，对锁的计数减1，更新缓存中的数据，如果当前对象对应的锁计数为0了，直接将其从 **tls** 缓存中删除。

```
lockCount--;
tls_set_direct(SYNC_COUNT_DIRECT_KEY, (void*)lockCount);
if (lockCount == 0) {
    tls_set_direct(SYNC_DATA_DIRECT_KEY, NULL);
}
```

OC

OSAtomicDecrement32Barri

2. 从线程缓存 **SyncCache** 中查找，如果找到，对锁的计数减1，更新缓存中的数据，如果当前，对应的锁计数为0了 直接将其从线程缓存 **SyncCache** 中删除。


[首页](#)
[探索掘金](#)




```
item->lockCount--;  
if (item->lockCount == 0) {  
    cache->list[i] = cache->list[--cache->used];  
}                                OSAtomicDecrement32Barrier(
```

3. 从 `sDataLists` 查找，找到的话，直接将其置为 `nil`。

其实，`@synchronized` 就是一个递归锁，其内部维护了一张表用来存储对象和锁的相关信息，加锁和释放锁的操作就是对锁的计数进行操作。

## 注意点

使用 `@synchronized` 的需要注意的是

```
for (int i = 0; i < 200000; i++) {  
    dispatch_async(dispatch_get_global_queue(0, 0), ^{  
        self.mArray = [NSMutableArray array];  
    });  
}
```

这段代码运行就会崩溃，是因为，我们在不断地创建 `array`，`mArray` 在不断的赋新值，释放旧值，这个时候多线程操作就会可能存在值已经被释放了，而其他线程还在操作，此时就会发生崩溃。此时就需要我们对程序加锁。将上述程序改成如下：

```
@synchronized (self.mArray) {  
    self.mArray = [NSMutableArray array];  
}
```

程序依然会崩溃，原因是 `@synchronized` 的操作时如果是 `nil`，则什么也不做，则可能会出现 `锁不住` 的情况，同样会导致在释放的时候发现值已经变成 `nil` 了。那我们应该怎么改呢？

第一种方式就是使用信号量加锁：

```
dispatch_semaphore_wait(_semp, DISPATCH_TIME_FOREVER);  
dispatch_async(dispatch_get_global_queue(0, 0), ^{  
    self.mArray = [NSMutableArray array];  
    dispatch_semaphore_signal(self.semp);  
});
```



## 第二种直接使用 `NSLock` :

```
NSLock *lock = [[NSLock alloc] init];
for (int i = 0; i < 200000; i++) {
    dispatch_async(dispatch_get_global_queue(0, 0), ^{
        [lock lock];
        self.mArray = [NSMutableArray array];
        [lock unlock];
    });
}
```

0C

在平常的开发中我们要慎用 `@synchronized(self)`，直接将 `self` 传入 `@synchronized` 确实是很简单粗暴的方法，但是这样容易导致死锁的出现。原因是因为 `self` 很可能被外部对象访问，被用作 `key` 来生成锁。两个公共锁交替使用的场景就容易出现死锁。

## 总结

`@synchronized` 是递归锁，其实是在底层对 `recursive_mutex_t` 做了封装和特殊处理。

让 `@synchronized` 具备处理递归能力的是 `lockCount`，让其能够处理多线程的是 `threadCount`。

进入代码块的入口是 `objc_sync_enter(id obj)`，出口是 `objc_sync_exit(id obj)`。

核心的处理如下：

- 如果支持 `tls` 缓存，就从 `tls` 缓存中查找对象锁 `SyncData`，找到对 `lockCount` 进行相应的操作
- 如果不支持 `tls` 缓存，或者从 `tls` 缓存中未找到，就从线程缓存 `SyncCache` 中查找，同样，找到对 `lockCount` 进行相应的操作
- 如果没有缓存命中，就从 `sDataLists` 链表中查找，找到之后进行相关的操作，并写入 `tls` 缓存和线程缓存 `SyncCache`
- 都没有找到，就创建一个节点，将对象锁 `SyncData` 插入 `sDataLists`，并写入缓存

释放对象操作类似。

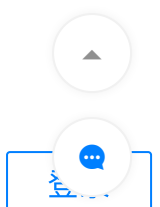
需要注意的是 `@synchronized` 的操作相对其他锁来说对性能消耗比较大，不建议大量使用。另外再某些多线程操作中，`@synchronized` 可能存在锁不住的情况。


关注下面的标签 发现更多相似内容



首页 ▾

探索掘金





**xcgg** Lv1 程序员

获得点赞 29 · 获得阅读 3,851

关注

安装掘金浏览器插件

打开新标签页发现好内容，掘金、GitHub、Dribbble、ProductHunt 等站点内容轻松获取。快来安装掘金浏览器插件获取高质量内容吧！

输入评论...

相关推荐

橘子不酸丶 · 2天前 · iOS  
**iOS优化篇之App启动时间优化**

👍 32

💬 1

老司机技术周报 · 1天前 · iOS / SwiftUI  
**【WWDC20】10037 - SwiftUI 中的 App 要领**

👍 8

💬

FengyunSky · 4天前 · iOS  
**一文读懂iOS线程调用栈原理**

👍 12

💬

ZenonHuang · 7天前 · iOS  
**iOS 的自动构建流程**

👍 61

💬 7

JeremyHuang37 · 1天前 · iOS  
**【译】自定义 Collection View Layout -- 一个简单的模板**

👍

💬 2

掘金酱 · 27天前 · iOS / Android / 前端 / 后端 / 程序员  
**🏆 掘金征文 | 2020与我的年中总结**

Chouee · 3天前 · iOS

造轮子 - UITableView字母索引条

 7

 1

路过看风景 · 2天前 · iOS


CocoaPods原理 及 组件化

 3



阿里巴巴淘系技术 · 5天前 · iOS

Apple Widget：下一个顶级流量入口？

 9



路过看风景 · 1天前 · iOS

iOS事件处理 UIResponder

 2



