GCD 源码分析

> 逻辑教育：知识从未如此性感，学习从未如此轻松
> 更多资料： 小雁子老师： 1900009930

0x01 Mach

Mach是XNU的核心，被BSD层包装。XNU由以下几个组件组成：

- MACH内核
  - 进程和线程抽象
  - 虚拟内存管理
  - 任务调度
  - 进程间通信和消息传递机制
- BSD
  - UNIX进程模型
  - POSIX线程模型
  - UNIX用户与组
  - 网络协议栈
  - 文件系统访问
  - 设备访问
- libKern
- I/O Kit

Mach的独特之处在于选择了通过消息传递的方式实现对象与对象之间的通信。而其他架构一个对象要访问另一个对象需要通过一个大家都知道的接口，而Mach对象不能直接调用另一个对象，而是必须传递消息。

一条消息就像网络包一样，定义为透明的blob(binary larger object，二进制大对象)，通过固定的包头进行分装

```
typedef struct
{
        mach_msg_header_t       header;
        mach_msg_body_t         body;
} mach_msg_base_t;

typedef struct
{
  mach_msg_bits_t   msgh_bits; // 消息头标志位
```

```
    mach_msg_size_t   msgh_size; // 大小
    mach_port_t       msgh_remote_port; // 目标(发消息)或源(接消息)
    mach_port_t       msgh_local_port; // 源(发消息)或目标(接消息)
    mach_port_name_t  msgh_voucher_port;
    mach_msg_id_t     msgh_id; // 唯一id
 } mach_msg_header_t;
```

Mach消息的发送和接收都是通过同一个API函数mach_msg()进行的。这个函数在用户态和内核态都有实现。为了实现消息的发送和接收，mach_msg()函数调用了一个Mach陷阱(trap)。Mach陷阱就是Mach中和系统调用等同的概念。在用户态调用mach_msg_trap()会引发陷阱机制，切换到内核态，在内核态中，内核实现的mach_msg()会完成实际的工作。这个函数也将会在下面的源码分析中遇到。

每一个BSD进程都在底层关联一个Mach任务对象，因为Mach提供的都是非常底层的抽象，提供的API从设计上讲很基础且不完整，所以需要在这之上提供一个更高的层次以实现完整的功能。我们开发层遇到的进程和线程就是BSD层对Mach的任务和线程的复杂包装。

进程填充的是线程，而线程是二进制代码的实际执行单元。用户态的线程始于对pthread_create的调用。这个函数的又由bsdthread_create系统调用完成，而bsdthread_create又其实是Mach中的thread_create的复杂包装，说到底真正的线程创建还是有Mach层完成。

在UNIX中，进程不能被创建出来，都是通过fork()系统调用复制出来的。复制出来的进程都会被要加载的执行程序覆盖整个内存空间。

接着，了解下常用的宏和常用的数据结构体。

0x02 源码中常见的宏

`1. __builtin_expect`

这个其实是个函数，针对编译器优化的一个函数，后面几个宏是对这个函数的封装，所以提前拎出来说一下。写代码中我们经常会遇到条件判断语句

```
if(今天是工作日) {
printf("好好上班");
}else{
printf("好好睡觉");
}
```

CPU读取指令的时候并非一条一条的来读，而是多条一起加载进来，比如已经加载了if(今天是工作日) printf(“好好上班”);的指令，这时候条件式如果为非，也就是非工作日，那么CPU继续把printf(“好好睡觉”);这条指令加载进来，这样就造成了性能浪费的现象。

__builtin_expect的第一个参数是实际值，第二个参数是预测值。使用这个目的是告诉编译器if条件式是不是有更大的可能被满足。

2. likely和unlikely

解开这个宏后其实是对__builtin_expect封装，likely表示更大可能成立，unlikely表示更大可能不成立。

```
#define likely(x) __builtin_expect(!!(x), 1)
#define unlikely(x) __builtin_expect(!!(x), 0)
```

遇到这样的,if(likely(a == 0))理解成if(a==0)即可，unlikely也是同样的。

3. fastpath和slowpath

跟上面也是差不多的，fastpath表示更大可能成立，slowpath表示更大可能不成立

```
#define fastpath(x) ((typeof(x))__builtin_expect(_safe_cast_to_long(x), ~0l))
#define slowpath(x) ((typeof(x))__builtin_expect(_safe_cast_to_long(x), 0l))
```

这两个理解起来跟likely和unlikely一样，只需要关注里面的条件式是否满足即可。

4. os_atomic_cmpxchg

其内部就是atomic_compare_exchange_strong_explicit函数，这个函数的作用是：第二个参数与第一个参数值比较，如果相等，第三个参数的值替换第一个参数的值。如果不相等，把第一个参数的值赋值到第二个参数上。

```
#define os_atomic_cmpxchg(p, e, v, m) \
        ({ _os_atomic_basetypeof(p) _r = (e); \
        atomic_compare_exchange_strong_explicit(_os_atomic_c11_atomic(p), \
        &_r, v, memory_order_##m, memory_order_relaxed); })
```

5. os_atomic_store2o

将第二个参数，保存到第一个参数

```
#define os_atomic_store2o(p, f, v, m)  os_atomic_store(&(p)->f, (v), m)
#define os_atomic_store(p, v, m) \
      atomic_store_explicit(_os_atomic_c11_atomic(p), v, memory_order_##m)
```

6. os_atomic_inc_orig

将1保存到第一个参数中

```
#define os_atomic_inc_orig(p, m)  os_atomic_add_orig((p), 1, m)
#define os_atomic_add_orig(p, v, m) _os_atomic_c11_op_orig((p), (v), m, add,
+)
#define _os_atomic_c11_op_orig(p, v, m, o, op) \
        atomic_fetch_##o##_explicit(_os_atomic_c11_atomic(p), v, \
        memory_order_##m)
```

0x03 数据结构体

接着，了解一些常用数据结构体。

1. dispatch_queue_t

```
typedef struct dispatch_queue_s *dispatch_queue_t;

  我们看下dispatch_queue_s怎么定义的。发现其内部有个_DISPATCH_QUEUE_HEADER宏定义。

struct dispatch_queue_s {
    _DISPATCH_QUEUE_HEADER(queue);
    DISPATCH_QUEUE_CACHELINE_PADDING;
} DISPATCH_ATOMIC64_ALIGN;

  解开_DISPATCH_QUEUE_HEADER后发现又一个DISPATCH_OBJECT_HEADER宏定义，继续拆解

#define _DISPATCH_QUEUE_HEADER(x) \
    struct os_mpsc_queue_s _as_oq[0]; \
    DISPATCH_OBJECT_HEADER(x); \
    _OS_MPSC_QUEUE_FIELDS(dq, dq_state); \
    uint32_t dq_side_suspend_cnt; \
    dispatch_unfair_lock_s dq_sidelock; \
    union { \
        dispatch_queue_t dq_specific_q; \
        struct dispatch_source_refs_s *ds_refs; \
        struct dispatch_timer_source_refs_s *ds_timer_refs; \
        struct dispatch_mach_recv_refs_s *dm_recv_refs; \
    }; \
    DISPATCH_UNION_LE(uint32_t volatile dq_atomic_flags, \
```

```
        const uint16_t dq_width, \
        const uint16_t __dq_opaque \
    ); \
    DISPATCH_INTROSPECTION_QUEUE_HEADER

  还有一层宏_DISPATCH_OBJECT_HEADER

#define DISPATCH_OBJECT_HEADER(x) \
    struct dispatch_object_s _as_do[0]; \
    _DISPATCH_OBJECT_HEADER(x)
```

不熟悉##的作用的同学，这里先说明下这个作用就拼接成字符串，比如x为group的话，下面就会拼接为dispatch_group这样的。

```
#define _DISPATCH_OBJECT_HEADER(x) \
    struct _os_object_s _as_os_obj[0]; \
    OS_OBJECT_STRUCT_HEADER(dispatch_##x); \
    struct dispatch_##x##_s *volatile do_next; \
    struct dispatch_queue_s *do_targetq; \
    void *do_ctxt; \
    void *do_finalizer
```

来到OS_OBJECT_STRUCT_HEADER之后，我们需要注意一个成员变量，记住这个成员变量名字叫做do_vtable。再继续拆解_OS_OBJECT_HEADER发现里面起就是一个isa指针和引用计数一些信息。

```
#define OS_OBJECT_STRUCT_HEADER(x) \
    _OS_OBJECT_HEADER(\
    const void *_objc_isa, \
    do_ref_cnt, \
    do_xref_cnt); \
    // 注意这个成员变量，后面将任务Push到队列就是通过这个变量
    const struct x##_vtable_s *do_vtable

#define _OS_OBJECT_HEADER(isa, ref_cnt, xref_cnt) \
        isa; /* must be pointer-sized */ \
        int volatile ref_cnt; \
        int volatile xref_cnt
```

2. dispatch_continuation_t

说到这个结构体，如果没看过源码的话，肯定对这个结构体很陌生，因为对外的api里面没有跟continuation有关的。所以这里先说下这个结构体就是用来封装block对象的，保存block的上

下文环境和block执行函数等。

```
typedef struct dispatch_continuation_s {
    struct dispatch_object_s _as_do[0];
    DISPATCH_CONTINUATION_HEADER(continuation);
} *dispatch_continuation_t;
```

看下里面的宏:DISPATCH_CONTINUATION_HEADER

```
#define DISPATCH_CONTINUATION_HEADER(x) \
    union { \
        const void *do_vtable; \
        uintptr_t dc_flags; \
    }; \
    union { \
        pthread_priority_t dc_priority; \
        int dc_cache_cnt; \
        uintptr_t dc_pad; \
    }; \
    struct dispatch_##x##_s *volatile do_next; \
    struct voucher_s *dc_voucher; \
    dispatch_function_t dc_func; \
    void *dc_ctxt; \
    void *dc_data; \
    void *dc_other
```

3. dispatch_object_t

```
typedef union {
    struct _os_object_s *_os_obj;
    struct dispatch_object_s *_do;
    struct dispatch_continuation_s *_dc;
    struct dispatch_queue_s *_dq;
    struct dispatch_queue_attr_s *_dqa;
    struct dispatch_group_s *_dg;
    struct dispatch_source_s *_ds;
    struct dispatch_mach_s *_dm;
    struct dispatch_mach_msg_s *_dmsg;
    struct dispatch_source_attr_s *_dsa;
    struct dispatch_semaphore_s *_dsema;
    struct dispatch_data_s *_ddata;
    struct dispatch_io_s *_dchannel;
    struct dispatch_operation_s *_doperation;
```

```
    struct dispatch_disk_s *_ddisk;
} dispatch_object_t DISPATCH_TRANSPARENT_UNION;
```

4. dispatch_function_t

dispatch_function_t 只是一个函数指针

```
typedef void (*dispatch_function_t)(void *_Nullable);
```

至此，一些常用的宏和数据结构体介绍完毕，接下来，我们真正的要一起阅读GCD相关的源码了。

0x03 创建队列

首先我们先从创建队列讲起。我们已经很熟悉，创建队列的方法是调用dispatch_queue_create函数。

```
    其内部又调用了_dispatch_queue_create_with_target函数
    DISPATCH_TARGET_QUEUE_DEFAULT这个宏其实就是null

    dispatch_queue_t dispatch_queue_create(const char *label, dispatch_queue_
attr_t attr)
    {   // attr一般我们都是传DISPATCH_QUEUE_SERIAL、DISPATCH_QUEUE_CONCURRENT或
者nil
        // 而DISPATCH_QUEUE_SERIAL其实就是null
        return _dispatch_queue_create_with_target(label, attr,
                DISPATCH_TARGET_QUEUE_DEFAULT, true);
    }
```

_dispatch_queue_create_with_target函数，这里会创建一个root队列,并将自己新建的队列绑定到所对应的root队列上。

```
static dispatch_queue_t _dispatch_queue_create_with_target(const char *label,
 dispatch_queue_attr_t dqa,
        dispatch_queue_t tq, bool legacy)
{   // 根据上文代码注释里提到的，作者认为调用者传入DISPATCH_QUEUE_SERIAL和nil的几率要
大于传DISPATCH_QUEUE_CONCURRENT。所以这里设置个默认值。
    // 这里怎么理解呢？只要看做if(!dqa)即可
    if (!slowpath(dqa)) {
        // _dispatch_get_default_queue_attr里面会将dqa的dqa_autorelease_frequen
cy指定为DISPATCH_AUTORELEASE_FREQUENCY_INHERIT的，inactive也指定为false。这里就不
展开了，只需要知道赋了哪些值。因为后面会用到。
        dqa = _dispatch_get_default_queue_attr();
```

```
    } else if (dqa->do_vtable != DISPATCH_VTABLE(queue_attr)) {
        DISPATCH_CLIENT_CRASH(dqa->do_vtable, "Invalid queue attribute");
    }

    // 取出优先级
    dispatch_qos_t qos = _dispatch_priority_qos(dqa->dqa_qos_and_relpri);

    // overcommit单纯从英文理解表示过量使用的意思，那这里这个overcommit就是一个标识符，
表示是不是就算负荷很高了，但还是得给我新开一个线程出来给我执行任务。
    _dispatch_queue_attr_overcommit_t overcommit = dqa->dqa_overcommit;
    if (overcommit != _dispatch_queue_attr_overcommit_unspecified && tq) {
        if (tq->do_targetq) {
            DISPATCH_CLIENT_CRASH(tq, "Cannot specify both overcommit and "
                    "a non-global target queue");
        }
    }

    // 如果overcommit没有被指定
    if (overcommit == _dispatch_queue_attr_overcommit_unspecified) {
         // 所以对于overcommit，如果是串行的话默认是开启的，而并行是关闭的
        overcommit = dqa->dqa_concurrent ?
                _dispatch_queue_attr_overcommit_disabled :
                _dispatch_queue_attr_overcommit_enabled;
    }

    // 之前说过初始化队列默认传了DISPATCH_TARGET_QUEUE_DEFAULT，也就是null，所以进入
if语句。
    if (!tq) {
        // 获取一个管理自己队列的root队列。
        tq = _dispatch_get_root_queue(
                qos == DISPATCH_QOS_UNSPECIFIED ? DISPATCH_QOS_DEFAULT : qos,
                overcommit == _dispatch_queue_attr_overcommit_enabled);
        if (slowpath(!tq)) {
            DISPATCH_CLIENT_CRASH(qos, "Invalid queue attribute");
        }
    }

    // legacy默认是true的
    if (legacy) {
        // 之前说过，默认是会给dqa_autorelease_frequency指定为DISPATCH_AUTORELEAS
E_FREQUENCY_INHERIT，所以这个判断式是成立的
        if (dqa->dqa_inactive || dqa->dqa_autorelease_frequency) {
            legacy = false;
        }
    }
```

```
    // vtable变量很重要，之后会被赋值到之前说的dispatch_queue_t结构体里的do_vtable变量上
    const void *vtable;
    dispatch_queue_flags_t dqf = 0;

    // legacy变为false了
    if (legacy) {
        vtable = DISPATCH_VTABLE(queue);
    } else if (dqa->dqa_concurrent) {
        // 如果创建队列的时候传了DISPATCH_QUEUE_CONCURRENT，就是走这里
        vtable = DISPATCH_VTABLE(queue_concurrent);
    } else {
        // 如果创建线程没有指定为并行队列，无论你传DISPATCH_QUEUE_SERIAL还是nil，都会创建一个串行队列。
        vtable = DISPATCH_VTABLE(queue_serial);
    }

    if (label) {
        // 判断传进来的字符串是否可变的，如果可变的copy成一份不可变的
        const char *tmp = _dispatch_strdup_if_mutable(label);
        if (tmp != label) {
            dqf |= DQF_LABEL_NEEDS_FREE;
            label = tmp;
        }
    }

    // _dispatch_object_alloc里面就将vtable赋值给do_vtable变量上了。
    dispatch_queue_t dq = _dispatch_object_alloc(vtable,
            sizeof(struct dispatch_queue_s) - DISPATCH_QUEUE_CACHELINE_PAD);
    // 第三个参数根据是否并行队列，如果不是则最多开一个线程，如果是则最多开0x1000 - 2个线程，这个数量很惊人了已经,换成十进制就是（4096 - 2）个。
    // dqa_inactive之前说串行是false的
    // DISPATCH_QUEUE_ROLE_INNER 也是0，所以这里串行队列的话dqa->dqa_state是0
    _dispatch_queue_init(dq, dqf, dqa->dqa_concurrent ?
            DISPATCH_QUEUE_WIDTH_MAX : 1, DISPATCH_QUEUE_ROLE_INNER |
            (dqa->dqa_inactive ? DISPATCH_QUEUE_INACTIVE : 0));

    dq->dq_label = label;
#if HAVE_PTHREAD_WORKQUEUE_QOS
    dq->dq_priority = dqa->dqa_qos_and_relpri;
    if (overcommit == _dispatch_queue_attr_overcommit_enabled) {
        dq->dq_priority |= DISPATCH_PRIORITY_FLAG_OVERCOMMIT;
    }
#endif
    _dispatch_retain(tq);
    if (qos == QOS_CLASS_UNSPECIFIED) {
```

```
        _dispatch_queue_priority_inherit_from_target(dq, tq);
    }
    if (!dqa->dqa_inactive) {
        _dispatch_queue_inherit_wlh_from_target(dq, tq);
    }
    // 自定义的queue的目标队列是root队列
    dq->do_targetq = tq;
    _dispatch_object_debug(dq, "%s", __func__);
    return _dispatch_introspection_queue_create(dq);
}
```

这个函数里面还是有几个重要的地方拆出来看下，首先是创建一个root队列_dispatch_get_root_queue函数。取root队列，一般是从一个装有12个root队列数组里面取。

```
static inline dispatch_queue_t
_dispatch_get_root_queue(dispatch_qos_t qos, bool overcommit)
{
    if (unlikely(qos == DISPATCH_QOS_UNSPECIFIED || qos > DISPATCH_QOS_MAX))
{
        DISPATCH_CLIENT_CRASH(qos, "Corrupted priority");
    }
    return &_dispatch_root_queues[2 * (qos - 1) + overcommit];
}
```

看下这个_dispatch_root_queues数组。我们可以看到，每一个优先级都有对应的root队列，每一个优先级又分为是不是可以过载的队列。

```
struct dispatch_queue_s _dispatch_root_queues[] = {
#define _DISPATCH_ROOT_QUEUE_IDX(n, flags) \
    ((flags & DISPATCH_PRIORITY_FLAG_OVERCOMMIT) ? \
        DISPATCH_ROOT_QUEUE_IDX_##n##_QOS_OVERCOMMIT : \
        DISPATCH_ROOT_QUEUE_IDX_##n##_QOS)
#define _DISPATCH_ROOT_QUEUE_ENTRY(n, flags, ...) \
    [_DISPATCH_ROOT_QUEUE_IDX(n, flags)] = { \
        DISPATCH_GLOBAL_OBJECT_HEADER(queue_root), \
        .dq_state = DISPATCH_ROOT_QUEUE_STATE_INIT_VALUE, \
        .do_ctxt = &_dispatch_root_queue_contexts[ \
                _DISPATCH_ROOT_QUEUE_IDX(n, flags)], \
        .dq_atomic_flags = DQF_WIDTH(DISPATCH_QUEUE_WIDTH_POOL), \
        .dq_priority = _dispatch_priority_make(DISPATCH_QOS_##n, 0) | flags |
 \
                DISPATCH_PRIORITY_FLAG_ROOTQUEUE | \
                ((flags & DISPATCH_PRIORITY_FLAG_DEFAULTQUEUE) ? 0 : \
                DISPATCH_QOS_##n << DISPATCH_PRIORITY_OVERRIDE_SHIFT), \
```

```
        __VA_ARGS__ \
    }
    _DISPATCH_ROOT_QUEUE_ENTRY(MAINTENANCE, 0,
        .dq_label = "com.apple.root.maintenance-qos",
        .dq_serialnum = 4,
    ),
    _DISPATCH_ROOT_QUEUE_ENTRY(MAINTENANCE, DISPATCH_PRIORITY_FLAG_OVERCOMMIT
,
        .dq_label = "com.apple.root.maintenance-qos.overcommit",
        .dq_serialnum = 5,
    ),
    _DISPATCH_ROOT_QUEUE_ENTRY(BACKGROUND, 0,
        .dq_label = "com.apple.root.background-qos",
        .dq_serialnum = 6,
    ),
    _DISPATCH_ROOT_QUEUE_ENTRY(BACKGROUND, DISPATCH_PRIORITY_FLAG_OVERCOMMIT,
        .dq_label = "com.apple.root.background-qos.overcommit",
        .dq_serialnum = 7,
    ),
    _DISPATCH_ROOT_QUEUE_ENTRY(UTILITY, 0,
        .dq_label = "com.apple.root.utility-qos",
        .dq_serialnum = 8,
    ),
    _DISPATCH_ROOT_QUEUE_ENTRY(UTILITY, DISPATCH_PRIORITY_FLAG_OVERCOMMIT,
        .dq_label = "com.apple.root.utility-qos.overcommit",
        .dq_serialnum = 9,
    ),
    _DISPATCH_ROOT_QUEUE_ENTRY(DEFAULT, DISPATCH_PRIORITY_FLAG_DEFAULTQUEUE,
        .dq_label = "com.apple.root.default-qos",
        .dq_serialnum = 10,
    ),
    _DISPATCH_ROOT_QUEUE_ENTRY(DEFAULT,
            DISPATCH_PRIORITY_FLAG_DEFAULTQUEUE | DISPATCH_PRIORITY_FLAG_OVER
COMMIT,
        .dq_label = "com.apple.root.default-qos.overcommit",
        .dq_serialnum = 11,
    ),
    _DISPATCH_ROOT_QUEUE_ENTRY(USER_INITIATED, 0,
        .dq_label = "com.apple.root.user-initiated-qos",
        .dq_serialnum = 12,
    ),
    _DISPATCH_ROOT_QUEUE_ENTRY(USER_INITIATED, DISPATCH_PRIORITY_FLAG_OVERCOM
MIT,
        .dq_label = "com.apple.root.user-initiated-qos.overcommit",
        .dq_serialnum = 13,
    ),
```

```
    _DISPATCH_ROOT_QUEUE_ENTRY(USER_INTERACTIVE, 0,
        .dq_label = "com.apple.root.user-interactive-qos",
        .dq_serialnum = 14,
    ),
    _DISPATCH_ROOT_QUEUE_ENTRY(USER_INTERACTIVE, DISPATCH_PRIORITY_FLAG_OVERC
OMMIT,
        .dq_label = "com.apple.root.user-interactive-qos.overcommit",
        .dq_serialnum = 15,
    ),
};
```

其中DISPATCH_GLOBAL_OBJECT_HEADER(queue_root)，解析到最后是OSdispatch##name##_class这样的这样的，对应的实例对象是如下代码，指定了root队列各个操作对应的函数。

```
DISPATCH_VTABLE_SUBCLASS_INSTANCE(queue_root, queue,
    .do_type = DISPATCH_QUEUE_GLOBAL_ROOT_TYPE,
    .do_kind = "global-queue",
    .do_dispose = _dispatch_pthread_root_queue_dispose,
    .do_push = _dispatch_root_queue_push,
    .do_invoke = NULL,
    .do_wakeup = _dispatch_root_queue_wakeup,
    .do_debug = dispatch_queue_debug,
);
```

其次看下DISPATCH_VTABLE这个宏，这个宏很重要。最后解封也是&OSdispatch##name##_class这样的。其实就是取dispatch_object_t对象。

如下代码，这里再举个VTABLE的串行对象，里面有各个状态该执行的函数：销毁函、挂起、恢复、push等函数都是在这里指定的。所以这里的do_push我们需要特别留意，后面push block任务到队列，就是通过调用do_push。

```
DISPATCH_VTABLE_SUBCLASS_INSTANCE(queue_serial, queue,
    .do_type = DISPATCH_QUEUE_SERIAL_TYPE,
    .do_kind = "serial-queue",
    .do_dispose = _dispatch_queue_dispose,
    .do_suspend = _dispatch_queue_suspend,
    .do_resume = _dispatch_queue_resume,
    .do_finalize_activation = _dispatch_queue_finalize_activation,
    .do_push = _dispatch_queue_push,
    .do_invoke = _dispatch_queue_invoke,
    .do_wakeup = _dispatch_queue_wakeup,
    .do_debug = dispatch_queue_debug,
    .do_set_targetq = _dispatch_queue_set_target_queue,
);
```

继续看下_dispatch_object_alloc和_dispatch_queue_init两个函数，首先看下_dispatch_object_alloc函数

```
void * _dispatch_object_alloc(const void *vtable, size_t size)
{
// OS_OBJECT_HAVE_OBJC1为1的满足式是：
// #if TARGET_OS_MAC && !TARGET_OS_SIMULATOR && defined(__i386__)
// 所以对于iOS并不满足
#if OS_OBJECT_HAVE_OBJC1
    const struct dispatch_object_vtable_s *_vtable = vtable;
    dispatch_object_t dou;
    dou._os_obj = _os_object_alloc_realized(_vtable->_os_obj_objc_isa, size);
    dou._do->do_vtable = vtable;
    return dou._do;
#else
    return _os_object_alloc_realized(vtable, size);
#endif
}

inline _os_object_t _os_object_alloc_realized(const void *cls, size_t size)
{
    _os_object_t obj;
    dispatch_assert(size >= sizeof(struct _os_object_s));
    while (!fastpath(obj = calloc(1u, size))) {
        _dispatch_temporary_resource_shortage();
    }
    obj->os_obj_isa = cls;
    return obj;
}

void _dispatch_temporary_resource_shortage(void)
{
    sleep(1);
    asm("");  // prevent tailcall
}
```

再看下_dispatch_queue_init函数，这里也就是做些初始化工作了

```
static inline void _dispatch_queue_init(dispatch_queue_t dq, dispatch_queue_flags_t dqf,
        uint16_t width, uint64_t initial_state_bits)
{
    uint64_t dq_state = DISPATCH_QUEUE_STATE_INIT_VALUE(width);

    dispatch_assert((initial_state_bits & ~(DISPATCH_QUEUE_ROLE_MASK |
```

```
            DISPATCH_QUEUE_INACTIVE)) == 0);

    if (initial_state_bits & DISPATCH_QUEUE_INACTIVE) {
        dq_state |= DISPATCH_QUEUE_INACTIVE + DISPATCH_QUEUE_NEEDS_ACTIVATION
;
        dq_state |= DLOCK_OWNER_MASK;
        dq->do_ref_cnt += 2;
    }

    dq_state |= (initial_state_bits & DISPATCH_QUEUE_ROLE_MASK);
    // 指向DISPATCH_OBJECT_LISTLESS是优化编译器的作用。只是为了生成更好的指令让CPU更好
的编码
    dq->do_next = (struct dispatch_queue_s *)DISPATCH_OBJECT_LISTLESS;
    dqf |= DQF_WIDTH(width);
    // dqf 保存进 dq->dq_atomic_flags
    os_atomic_store2o(dq, dq_atomic_flags, dqf, relaxed);
    dq->dq_state = dq_state;
    dq->dq_serialnum =
            os_atomic_inc_orig(&_dispatch_queue_serial_numbers, relaxed);
}
```

最后是_dispatch_introspection_queue_create函数，一个内省函数。

```
dispatch_queue_t _dispatch_introspection_queue_create(dispatch_queue_t dq)
{
    TAILQ_INIT(&dq->diq_order_top_head);
    TAILQ_INIT(&dq->diq_order_bottom_head);
    _dispatch_unfair_lock_lock(&_dispatch_introspection.queues_lock);
    TAILQ_INSERT_TAIL(&_dispatch_introspection.queues, dq, diq_list);
    _dispatch_unfair_lock_unlock(&_dispatch_introspection.queues_lock);

    DISPATCH_INTROSPECTION_INTERPOSABLE_HOOK_CALLOUT(queue_create, dq);
    if (DISPATCH_INTROSPECTION_HOOK_ENABLED(queue_create)) {
        _dispatch_introspection_queue_create_hook(dq);
    }
    return dq;
}
```

至此，一个队列的创建过程我们大致了解了。大致可以分为这么几点

设置队列优先级
默认创建的是一个串行队列
设置队列挂载的根队列。优先级不同根队列也不同
实例化vtable对象，这个对象给不同队列指定了push、wakeup等函数。

0x04 dispatch_sync

dispatch_sync直接调用的是dispatch_sync_f

```
void dispatch_sync(dispatch_queue_t dq, dispatch_block_t work)
{
    // 很大可能不会走if分支，看做if(_dispatch_block_has_private_data(work))
    if (unlikely(_dispatch_block_has_private_data(work))) {
        return _dispatch_sync_block_with_private_data(dq, work, 0);
    }
    dispatch_sync_f(dq, work, _dispatch_Block_invoke(work));
}

void
dispatch_sync_f(dispatch_queue_t dq, void *ctxt, dispatch_function_t func)
{
    // 串行队列会走到这个if分支
    if (likely(dq->dq_width == 1)) {
        return dispatch_barrier_sync_f(dq, ctxt, func);
    }

    // 全局获取的并行队列或者绑定的是非调度线程的队列会走进这个if分支
    if (unlikely(!_dispatch_queue_try_reserve_sync_width(dq))) {
        return _dispatch_sync_f_slow(dq, ctxt, func, 0);
    }

    _dispatch_introspection_sync_begin(dq);
    if (unlikely(dq->do_targetq->do_targetq)) {
        return _dispatch_sync_recurse(dq, ctxt, func, 0);
    }
    // 自定义并行队列会来到这个函数
    _dispatch_sync_invoke_and_complete(dq, ctxt, func);
}
```

先说第一种情况，串行队列。

```
void dispatch_barrier_sync_f(dispatch_queue_t dq, void *ctxt,
        dispatch_function_t func)
{
    dispatch_tid tid = _dispatch_tid_self();

    // 队列绑定的是非调度线程就会走这里
    if (unlikely(!_dispatch_queue_try_acquire_barrier_sync(dq, tid))) {
        return _dispatch_sync_f_slow(dq, ctxt, func, DISPATCH_OBJ_BARRIER_BIT
```

```
);
    }

    _dispatch_introspection_sync_begin(dq);
    if (unlikely(dq->do_targetq->do_targetq)) {
        return _dispatch_sync_recurse(dq, ctxt, func, DISPATCH_OBJ_BARRIER_BI
T);
    }
    // 一般会走到这里
    _dispatch_queue_barrier_sync_invoke_and_complete(dq, ctxt, func);
}

static void _dispatch_queue_barrier_sync_invoke_and_complete(dispatch_queue_t
 dq,
        void *ctxt, dispatch_function_t func)
{
    // 首先会执行这个函数
    _dispatch_sync_function_invoke_inline(dq, ctxt, func);
    // 如果后面还有别的任务
    if (unlikely(dq->dq_items_tail || dq->dq_width > 1)) {
        // 内部其实就是唤醒队列
        return _dispatch_queue_barrier_complete(dq, 0, 0);
    }

    const uint64_t fail_unlock_mask = DISPATCH_QUEUE_SUSPEND_BITS_MASK |
            DISPATCH_QUEUE_ENQUEUED | DISPATCH_QUEUE_DIRTY |
            DISPATCH_QUEUE_RECEIVED_OVERRIDE | DISPATCH_QUEUE_SYNC_TRANSFER |
            DISPATCH_QUEUE_RECEIVED_SYNC_WAIT;
    uint64_t old_state, new_state;

    // 原子锁。检查dq->dq_state与old_state是否相等，如果相等把new_state赋值给dq->dq_
state，如果不相等，把dq_state赋值给old_state。
    // 串行队列走到这里，dq->dq_state与old_state是相等的，会把new_state也就是闭包里的
赋值的值给dq->dq_state
    os_atomic_rmw_loop2o(dq, dq_state, old_state, new_state, release, {
        new_state  = old_state - DISPATCH_QUEUE_SERIAL_DRAIN_OWNED;
        new_state &= ~DISPATCH_QUEUE_DRAIN_UNLOCK_MASK;
        new_state &= ~DISPATCH_QUEUE_MAX_QOS_MASK;
        if (unlikely(old_state & fail_unlock_mask)) {
            os_atomic_rmw_loop_give_up({
                return _dispatch_queue_barrier_complete(dq, 0, 0);
            });
        }
    });
    if (_dq_state_is_base_wlh(old_state)) {
        _dispatch_event_loop_assert_not_owned((dispatch_wlh_t)dq);
```

```
    }
}

static inline void _dispatch_sync_function_invoke_inline(dispatch_queue_t dq,
 void *ctxt,
        dispatch_function_t func)
{
    // 保护现场 -> 调用函数 -> 恢复现场
    dispatch_thread_frame_s dtf;
    _dispatch_thread_frame_push(&dtf, dq);
    _dispatch_client_callout(ctxt, func);
    _dispatch_perfmon_workitem_inc();
    _dispatch_thread_frame_pop(&dtf);
}
```

然后另一种情况，自定义并行队列会走_dispatch_sync_invoke_and_complete函数。

```
static void _dispatch_sync_invoke_and_complete(dispatch_queue_t dq, void *ctx
t,
        dispatch_function_t func)
{
    _dispatch_sync_function_invoke_inline(dq, ctxt, func);
    // 将自定义队列加入到root队列里去
    // dispatch_async也会调用此方法，之前我们初始化的时候会绑定一个root队列，这里就将我
们新建的队列交给root队列进行管理
    _dispatch_queue_non_barrier_complete(dq);
}

static inline void _dispatch_sync_function_invoke_inline(dispatch_queue_t dq,
 void *ctxt,
        dispatch_function_t func)
{
    dispatch_thread_frame_s dtf;
    _dispatch_thread_frame_push(&dtf, dq);
    // 执行任务
    _dispatch_client_callout(ctxt, func);
    _dispatch_perfmon_workitem_inc();
    _dispatch_thread_frame_pop(&dtf);
}
```

0x05 dispatch_async

内部就是两个函数_dispatch_continuation_init和_dispatch_continuation_async

```
void dispatch_async(dispatch_queue_t dq, dispatch_block_t work)
{
```

```
    dispatch_continuation_t dc = _dispatch_continuation_alloc();
    // 设置标识位
    uintptr_t dc_flags = DISPATCH_OBJ_CONSUME_BIT;

    _dispatch_continuation_init(dc, dq, work, 0, 0, dc_flags);
    _dispatch_continuation_async(dq, dc);
}
```

_dispatch_continuation_init函数只是一个初始化，主要就是保存Block上下文，指定block的执行函数

```
static inline void _dispatch_continuation_init(dispatch_continuation_t dc,
        dispatch_queue_class_t dqu, dispatch_block_t work,
        pthread_priority_t pp, dispatch_block_flags_t flags, uintptr_t dc_fla
gs)
{
    dc->dc_flags = dc_flags | DISPATCH_OBJ_BLOCK_BIT;
    // block对象赋值到dc_ctxt
    dc->dc_ctxt = _dispatch_Block_copy(work);
    // 设置默认任务优先级
    _dispatch_continuation_priority_set(dc, pp, flags);

    // 大多数情况不会走这个分支
    if (unlikely(_dispatch_block_has_private_data(work))) {
        return _dispatch_continuation_init_slow(dc, dqu, flags);
    }

    // 这个标识位多眼熟，就是前面入口赋值的，没的跑了，指定执行函数就是_dispatch_call_bl
ock_and_release了
    if (dc_flags & DISPATCH_OBJ_CONSUME_BIT) {
        dc->dc_func = _dispatch_call_block_and_release;
    } else {
        dc->dc_func = _dispatch_Block_invoke(work);
    }
    _dispatch_continuation_voucher_set(dc, dqu, flags);
}
```

_dispatch_call_block_and_release这个函数就是直接执行block了，所以dc->dc_func被调用的话就block会被直接执行了。

```
void _dispatch_call_block_and_release(void *block)
{
    void (^b)(void) = block;
    b();
    Block_release(b);
}
```

上面的初始化过程就是这样，接着看下_dispatch_continuation_async函数

```
void _dispatch_continuation_async(dispatch_queue_t dq, dispatch_continuation_
t dc)
{
    // 看看是不是barrier类型的block
    _dispatch_continuation_async2(dq, dc,
            dc->dc_flags & DISPATCH_OBJ_BARRIER_BIT);
}

static inline void _dispatch_continuation_async2(dispatch_queue_t dq, dispatc
h_continuation_t dc,
        bool barrier)
{
    // 如果是用barrier插进来的任务或者是串行队列，直接将任务加入到队列
    // #define DISPATCH_QUEUE_USES_REDIRECTION(width) \
    //    ({ uint16_t _width = (width); \
    //    _width > 1 && _width < DISPATCH_QUEUE_WIDTH_POOL; })
    if (fastpath(barrier || !DISPATCH_QUEUE_USES_REDIRECTION(dq->dq_width)))
{
        return _dispatch_continuation_push(dq, dc);
    }
    return _dispatch_async_f2(dq, dc);
}

// 可以先看下如果是barrier任务，直接调用_dispatch_continuation_push函数
static void _dispatch_continuation_push(dispatch_queue_t dq, dispatch_continu
ation_t dc)
{
    dx_push(dq, dc, _dispatch_continuation_override_qos(dq, dc));
}

// _dispatch_continuation_async2函数里面调用_dispatch_async_f2函数
static void
_dispatch_async_f2(dispatch_queue_t dq, dispatch_continuation_t dc)
{
    // 如果还有任务，slowpath表示很大可能队尾是没有任务的。
    // 实际开发中也的确如此，一般情况下我们不会dispatch_async之后又马上跟着一个dispatch
_async
    if (slowpath(dq->dq_items_tail)) {
        return _dispatch_continuation_push(dq, dc);
    }

    if (slowpath(!_dispatch_queue_try_acquire_async(dq))) {
        return _dispatch_continuation_push(dq, dc);
```

```
    }

    // 一般会直接来到这里，_dispatch_continuation_override_qos函数里面主要做的是判断dq有没有设置的优先级，如果没有就用block对象的优先级，如果有就用自己的
    return _dispatch_async_f_redirect(dq, dc,
            _dispatch_continuation_override_qos(dq, dc));
}

static void _dispatch_async_f_redirect(dispatch_queue_t dq,
        dispatch_object_t dou, dispatch_qos_t qos)
{
    // 这里会走进if的语句，因为_dispatch_object_is_redirection内部的dx_type(dou._do) == type条件为否
    if (!slowpath(_dispatch_object_is_redirection(dou))) {
        dou._dc = _dispatch_async_redirect_wrap(dq, dou);
    }
    // dq换成所绑定的root队列
    dq = dq->do_targetq;

    // 基本不会走里面的循环，主要做的就是找到根root队列
    while (slowpath(DISPATCH_QUEUE_USES_REDIRECTION(dq->dq_width))) {
        if (!fastpath(_dispatch_queue_try_acquire_async(dq))) {
            break;
        }
        if (!dou._dc->dc_ctxt) {
            dou._dc->dc_ctxt = (void *)
                    (uintptr_t)_dispatch_queue_autorelease_frequency(dq);
        }
        dq = dq->do_targetq;
    }

    // 把装有block信息的结构体装进所在队列对应的root_queue里面
    dx_push(dq, dou, qos);
}

// dx_push是个宏定义，这里做的就是将任务push到任务队列，我们看到这里，就知道dx_push就是调用对象的do_push。
#define dx_push(x, y, z) dx_vtable(x)->do_push(x, y, z)
#define dx_vtable(x) (&(x)->do_vtable->_os_obj_vtable)

  _dispatch_async_f_redirect函数里先看这句dou._dc = _dispatch_async_redirect_wrap(dq, dou);

static inline dispatch_continuation_t _dispatch_async_redirect_wrap(dispatch_queue_t dq, dispatch_object_t dou)
{
```

```
    dispatch_continuation_t dc = _dispatch_continuation_alloc();

    dou._do->do_next = NULL;
    // 所以dispatch_async推进的任务的do_vtable成员变量是有值的
    dc->do_vtable = DC_VTABLE(ASYNC_REDIRECT);
    dc->dc_func = NULL;
    dc->dc_ctxt = (void *)(uintptr_t)_dispatch_queue_autorelease_frequency(dq
);
    // 所属队列被装进dou._dc->dc_data里面了
    dc->dc_data = dq;
    dc->dc_other = dou._do;
    dc->dc_voucher = DISPATCH_NO_VOUCHER;
    dc->dc_priority = DISPATCH_NO_PRIORITY;
    _dispatch_retain(dq); // released in _dispatch_async_redirect_invoke
    return dc;
}

// dc->do_vtable = DC_VTABLE(ASYNC_REDIRECT); 就是下面指定redirect的invoke函数是
_dispatch_async_redirect_invoke，后面任务被执行就是通过这个函数
const struct dispatch_continuation_vtable_s _dispatch_continuation_vtables[]
= {
    DC_VTABLE_ENTRY(ASYNC_REDIRECT,
        .do_kind = "dc-redirect",
        .do_invoke = _dispatch_async_redirect_invoke),
#if HAVE_MACH
    DC_VTABLE_ENTRY(MACH_SEND_BARRRIER_DRAIN,
        .do_kind = "dc-mach-send-drain",
        .do_invoke = _dispatch_mach_send_barrier_drain_invoke),
    DC_VTABLE_ENTRY(MACH_SEND_BARRIER,
        .do_kind = "dc-mach-send-barrier",
        .do_invoke = _dispatch_mach_barrier_invoke),
    DC_VTABLE_ENTRY(MACH_RECV_BARRIER,
        .do_kind = "dc-mach-recv-barrier",
        .do_invoke = _dispatch_mach_barrier_invoke),
    DC_VTABLE_ENTRY(MACH_ASYNC_REPLY,
        .do_kind = "dc-mach-async-reply",
        .do_invoke = _dispatch_mach_msg_async_reply_invoke),
#endif
#if HAVE_PTHREAD_WORKQUEUE_QOS
    DC_VTABLE_ENTRY(OVERRIDE_STEALING,
        .do_kind = "dc-override-stealing",
        .do_invoke = _dispatch_queue_override_invoke),
    // 留意这个，后面也会被用到
    DC_VTABLE_ENTRY(OVERRIDE_OWNING,
        .do_kind = "dc-override-owning",
        .do_invoke = _dispatch_queue_override_invoke),
```

```
#endif
};
```

再看dx_push(dq, dou, qos);这句，其实就是调用_dispatch_root_queue_push函数

```
void _dispatch_root_queue_push(dispatch_queue_t rq, dispatch_object_t dou,
        dispatch_qos_t qos)
{
    // 一般情况下，无论自定义还是非自定义都会走进这个条件式(比如：dispatch_get_global_q
ueue)
    // 里面主要对比的是qos与root队列的qos是否一致。基本上都不一致的，如果不一致走进这个if
语句
    if (_dispatch_root_queue_push_needs_override(rq, qos)) {
        return _dispatch_root_queue_push_override(rq, dou, qos);
    }
    _dispatch_root_queue_push_inline(rq, dou, dou, 1);
}

static void _dispatch_root_queue_push_override(dispatch_queue_t orig_rq,
        dispatch_object_t dou, dispatch_qos_t qos)
{
    bool overcommit = orig_rq->dq_priority & DISPATCH_PRIORITY_FLAG_OVERCOMMI
T;
    dispatch_queue_t rq = _dispatch_get_root_queue(qos, overcommit);
    dispatch_continuation_t dc = dou._dc;
    // 这个_dispatch_object_is_redirection函数其实就是return _dispatch_object_ha
s_type(dou,DISPATCH_CONTINUATION_TYPE(ASYNC_REDIRECT));
    // 所以自定义队列会走这个if语句，如果是dispatch_get_global_queue不会走if语句
    if (_dispatch_object_is_redirection(dc)) {
        dc->dc_func = (void *)orig_rq;
    } else {
        // dispatch_get_global_queue来到这里
        dc = _dispatch_continuation_alloc();
        // 相当于是下面的，也就是指定了执行函数为_dispatch_queue_override_invoke，所
以有别于自定义队列的invoke函数。
        // DC_VTABLE_ENTRY(OVERRIDE_OWNING,
        // .do_kind = "dc-override-owning",
        // .do_invoke = _dispatch_queue_override_invoke),
        dc->do_vtable = DC_VTABLE(OVERRIDE_OWNING);
        _dispatch_trace_continuation_push(orig_rq, dou);
        dc->dc_ctxt = dc;
        dc->dc_other = orig_rq;
        dc->dc_data = dou._do;
        dc->dc_priority = DISPATCH_NO_PRIORITY;
        dc->dc_voucher = DISPATCH_NO_VOUCHER;
    }
```

```
    _dispatch_root_queue_push_inline(rq, dc, dc, 1);
}

static inline void _dispatch_root_queue_push_inline(dispatch_queue_t dq, disp
atch_object_t _head,
        dispatch_object_t _tail, int n)
{
    struct dispatch_object_s *head = _head._do, *tail = _tail._do;
    // 把任务装进队列，大多数不走进if语句。但是第一个任务进来之前还是满足这个条件式的，会进入这个条件语句去激活队列来执行里面的任务，后面再加入的任务因为队列被激活了，所以也就不太需要再进入这个队列了，所以相对来说激活队列只要一次，所以作者认为大多数情况下不需要走进这个条件语句
    if (unlikely(_dispatch_queue_push_update_tail_list(dq, head, tail))) {
        // 保存队列头
        _dispatch_queue_push_update_head(dq, head);
        return _dispatch_global_queue_poke(dq, n, 0);
    }
}
```

至此，我们可以看到，我们装入到自定义的任务都被扔到其挂靠的root队列里去了，所以我们我们自己创建的队列只是一个代理人身份，真正的管理人是其对应的root队列，但同时这个队列也是被管理的。

继续看_dispatch_global_queue_poke函数

```
void
_dispatch_global_queue_poke(dispatch_queue_t dq, int n, int floor)
{
    return _dispatch_global_queue_poke_slow(dq, n, floor);
}
```

继续看_dispatch_global_queue_poke函数调用了_dispatch_global_queue_poke_slow函数，这里也很关键了，里面执行_pthread_workqueue_addthreads函数，把任务交给内核分发处理

```
_dispatch_global_queue_poke_slow(dispatch_queue_t dq, int n, int floor)
{
    dispatch_root_queue_context_t qc = dq->do_ctxt;
    int remaining = n;
    int r = ENOSYS;

    _dispatch_root_queues_init();
    _dispatch_debug_root_queue(dq, __func__);
    if (qc->dgq_kworkqueue != (void*)(~0ul))
    {
        r = _pthread_workqueue_addthreads(remaining,
                _dispatch_priority_to_pp(dq->dq_priority));
```

```
        (void)dispatch_assume_zero(r);
        return;
    }
}

int
_pthread_workqueue_addthreads(int numthreads, pthread_priority_t priority)
{
    int res = 0;

    if (__libdispatch_workerfunction == NULL) {
        return EPERM;
    }

    if ((__pthread_supported_features & PTHREAD_FEATURE_FINEPRIO) == 0) {
        return ENOTSUP;
    }

    res = __workq_kernreturn(WQOPS_QUEUE_REQTHREADS, NULL, numthreads, (int)p
riority);
    if (res == -1) {
        res = errno;
    }
    return res;
}
```

那么，加入到根队列的任务是怎么被运行起来的？在此之前，我们先模拟一下在GCD内部把程序搞挂掉，这样我们就可以追溯下调用栈关系。

```
(
    0   CoreFoundation                      0x00000001093fe12b __exceptionPre
process + 171
    1   libobjc.A.dylib                     0x0000000108a92f41 objc_exception
_throw + 48
    2   CoreFoundation                      0x000000010943e0cc _CFThrowFormat
tedException + 194
    3   CoreFoundation                      0x000000010930c23d -[__NSPlacehol
derArray initWithObjects:count:] + 237
    4   CoreFoundation                      0x0000000109312e34 +[NSArray arra
yWithObjects:count:] + 52
    5   HotPatch                            0x000000010769df77 __29-[ViewCont
roller viewDidLoad]_block_invoke + 87
    6   libdispatch.dylib                   0x000000010c0a62f7 _dispatch_call
_block_and_release + 12
    7   libdispatch.dylib                   0x000000010c0a733d _dispatch_clie
```

```
nt_callout + 8
    8   libdispatch.dylib                   0x000000010c0ad754 _dispatch_cont
inuation_pop + 967
    9   libdispatch.dylib                   0x000000010c0abb85 _dispatch_asyn
c_redirect_invoke + 780
    10  libdispatch.dylib                   0x000000010c0b3102 _dispatch_root
_queue_drain + 772
    11  libdispatch.dylib                   0x000000010c0b2da0 _dispatch_work
er_thread3 + 132
    12  libsystem_pthread.dylib             0x000000010c5f95a2 _pthread_wqthr
ead + 1299
    13  libsystem_pthread.dylib             0x000000010c5f907d
    start_wqthread + 13
)
```

很明显，我们已经看到加入到队列的任务的调用关系是：

```
start_wqthread -> _pthread_wqthread -> _dispatch_worker_thread3 -> _dispatch_
root_queue_drain -> _dispatch_async_redirect_invoke -> _dispatch_continuation
_pop -> _dispatch_client_callout -> _dispatch_call_block_and_release
```

只看调用关系也不知道里面做了什么，所以还是上代码

```
// 根据优先级取出相应的root队列，再调用_dispatch_worker_thread4函数
static void _dispatch_worker_thread3(pthread_priority_t pp)
{
    bool overcommit = pp & _PTHREAD_PRIORITY_OVERCOMMIT_FLAG;
    dispatch_queue_t dq;
    pp &= _PTHREAD_PRIORITY_OVERCOMMIT_FLAG | ~_PTHREAD_PRIORITY_FLAGS_MASK;
    _dispatch_thread_setspecific(dispatch_priority_key, (void *)(uintptr_t)pp
);
    dq = _dispatch_get_root_queue(_dispatch_qos_from_pp(pp), overcommit);
    return _dispatch_worker_thread4(dq);
}
// 开始调用_dispatch_root_queue_drain函数，取出任务
static void _dispatch_worker_thread4(void *context)
{
    dispatch_queue_t dq = context;
    dispatch_root_queue_context_t qc = dq->do_ctxt;

    _dispatch_introspection_thread_add();
    int pending = os_atomic_dec2o(qc, dgq_pending, relaxed);
    dispatch_assert(pending >= 0);
    _dispatch_root_queue_drain(dq, _dispatch_get_priority());
```

```
    _dispatch_voucher_debug("root queue clear", NULL);
    _dispatch_reset_voucher(NULL, DISPATCH_THREAD_PARK);
}
// 循环取出任务
 static void _dispatch_root_queue_drain(dispatch_queue_t dq, pthread_priority
_t pp)
{
        _dispatch_queue_set_current(dq);
    dispatch_priority_t pri = dq->dq_priority;
    if (!pri) pri = _dispatch_priority_from_pp(pp);
    dispatch_priority_t old_dbp = _dispatch_set_basepri(pri);
    _dispatch_adopt_wlh_anon();

    struct dispatch_object_s *item;
    bool reset = false;
    dispatch_invoke_context_s dic = { };
        dispatch_invoke_flags_t flags = DISPATCH_INVOKE_WORKER_DRAIN |
            DISPATCH_INVOKE_REDIRECTING_DRAIN;
    _dispatch_queue_drain_init_narrowing_check_deadline(&dic, pri);
    _dispatch_perfmon_start();
    while ((item = fastpath(_dispatch_root_queue_drain_one(dq)))) {
        if (reset) _dispatch_wqthread_override_reset();
        _dispatch_continuation_pop_inline(item, &dic, flags, dq);
        reset = _dispatch_reset_basepri_override();
        if (unlikely(_dispatch_queue_drain_should_narrow(&dic))) {
            break;
        }
    }

    // overcommit or not. worker thread
    if (pri & _PTHREAD_PRIORITY_OVERCOMMIT_FLAG) {
        _dispatch_perfmon_end(perfmon_thread_worker_oc);
    } else {
        _dispatch_perfmon_end(perfmon_thread_worker_non_oc);
    }

    _dispatch_reset_wlh();
    _dispatch_reset_basepri(old_dbp);
    _dispatch_reset_basepri_override();
    _dispatch_queue_set_current(NULL);
}

// 这个函数的作用就是调度出任务的执行函数
static inline void _dispatch_continuation_pop_inline(dispatch_object_t dou,
        dispatch_invoke_context_t dic, dispatch_invoke_flags_t flags,
        dispatch_queue_t dq)
```

```
{
    dispatch_pthread_root_queue_observer_hooks_t observer_hooks =
            _dispatch_get_pthread_root_queue_observer_hooks();
    if (observer_hooks) observer_hooks->queue_will_execute(dq);
    _dispatch_trace_continuation_pop(dq, dou);
    flags &= _DISPATCH_INVOKE_PROPAGATE_MASK;

    // 之前说过dispatch_async是有do_vtable成员变量的，所以会走进这个if分支，又invoke
方法指定为_dispatch_async_redirect_invoke，所以执行该函数
    // 相同的，如果是dispatch_get_global_queue也会走这个分支，执行_dispatch_queue_o
verride_invoke方法，这个之前也说过了
    if (_dispatch_object_has_vtable(dou)) {
        dx_invoke(dou._do, dic, flags);
    } else {
        _dispatch_continuation_invoke_inline(dou, DISPATCH_NO_VOUCHER, flags)
;
    }
    if (observer_hooks) observer_hooks->queue_did_execute(dq);
}

// 继续按自定义队列的步骤走
void _dispatch_async_redirect_invoke(dispatch_continuation_t dc,
        dispatch_invoke_context_t dic, dispatch_invoke_flags_t flags)
{
    dispatch_thread_frame_s dtf;
    struct dispatch_continuation_s *other_dc = dc->dc_other;
    dispatch_invoke_flags_t ctxt_flags = (dispatch_invoke_flags_t)dc->dc_ctxt
;

    dispatch_queue_t assumed_rq = (dispatch_queue_t)dc->dc_func;
    dispatch_queue_t dq = dc->dc_data, rq, old_dq;
    dispatch_priority_t old_dbp;

    if (ctxt_flags) {
        flags &= ~_DISPATCH_INVOKE_AUTORELEASE_MASK;
        flags |= ctxt_flags;
    }
    old_dq = _dispatch_get_current_queue();
    if (assumed_rq) {
        old_dbp = _dispatch_root_queue_identity_assume(assumed_rq);
        _dispatch_set_basepri(dq->dq_priority);
    } else {
        old_dbp = _dispatch_set_basepri(dq->dq_priority);
    }

    _dispatch_thread_frame_push(&dtf, dq);
```

```
    // _dispatch_continuation_pop_forwarded里面就是执行_dispatch_continuation_p
op函数
    _dispatch_continuation_pop_forwarded(dc, DISPATCH_NO_VOUCHER,
            DISPATCH_OBJ_CONSUME_BIT, {
        _dispatch_continuation_pop(other_dc, dic, flags, dq);
    });
    _dispatch_thread_frame_pop(&dtf);
    if (assumed_rq) _dispatch_queue_set_current(old_dq);
    _dispatch_reset_basepri(old_dbp);

    rq = dq->do_targetq;
    while (slowpath(rq->do_targetq) && rq != old_dq) {
        _dispatch_queue_non_barrier_complete(rq);
        rq = rq->do_targetq;
    }

    _dispatch_queue_non_barrier_complete(dq);
    _dispatch_release_tailcall(dq);
}

// 顺便说下，如果按照的是dispatch_get_global_queue会执行_dispatch_queue_override_i
nvoke函数
void _dispatch_queue_override_invoke(dispatch_continuation_t dc,
        dispatch_invoke_context_t dic, dispatch_invoke_flags_t flags)
{
    dispatch_queue_t old_rq = _dispatch_queue_get_current();
    dispatch_queue_t assumed_rq = dc->dc_other;
    dispatch_priority_t old_dp;
    voucher_t ov = DISPATCH_NO_VOUCHER;
    dispatch_object_t dou;

    dou._do = dc->dc_data;
    old_dp = _dispatch_root_queue_identity_assume(assumed_rq);
    if (dc_type(dc) == DISPATCH_CONTINUATION_TYPE(OVERRIDE_STEALING)) {
        flags |= DISPATCH_INVOKE_STEALING;
    } else {
        // balance the fake continuation push in
        // _dispatch_root_queue_push_override
        _dispatch_trace_continuation_pop(assumed_rq, dou._do);
    }
    // 同样调用_dispatch_continuation_pop函数
    _dispatch_continuation_pop_forwarded(dc, ov, DISPATCH_OBJ_CONSUME_BIT, {
        if (_dispatch_object_has_vtable(dou._do)) {
            dx_invoke(dou._do, dic, flags);
        } else {
            _dispatch_continuation_invoke_inline(dou, ov, flags);
```

```
        }
    });
    _dispatch_reset_basepri(old_dp);
    _dispatch_queue_set_current(old_rq);
}
```

// 回归正题，无论是自定义的队列还是获取系统的，最终都会调用这个函数

```
void _dispatch_continuation_pop(dispatch_object_t dou, dispatch_invoke_contex
t_t dic,
        dispatch_invoke_flags_t flags, dispatch_queue_t dq)
{
    _dispatch_continuation_pop_inline(dou, dic, flags, dq);
}

static inline void _dispatch_continuation_pop_inline(dispatch_object_t dou,
        dispatch_invoke_context_t dic, dispatch_invoke_flags_t flags,
        dispatch_queue_t dq)
{
    dispatch_pthread_root_queue_observer_hooks_t observer_hooks =
            _dispatch_get_pthread_root_queue_observer_hooks();
    if (observer_hooks) observer_hooks->queue_will_execute(dq);
    _dispatch_trace_continuation_pop(dq, dou);
    flags &= _DISPATCH_INVOKE_PROPAGATE_MASK;
    if (_dispatch_object_has_vtable(dou)) {
        dx_invoke(dou._do, dic, flags);
    } else {
        _dispatch_continuation_invoke_inline(dou, DISPATCH_NO_VOUCHER, flags)
;
    }
    if (observer_hooks) observer_hooks->queue_did_execute(dq);
}

static inline void _dispatch_continuation_invoke_inline(dispatch_object_t dou
, voucher_t ov,
        dispatch_invoke_flags_t flags)
{
    dispatch_continuation_t dc = dou._dc, dc1;
    dispatch_invoke_with_autoreleasepool(flags, {
        uintptr_t dc_flags = dc->dc_flags;

        _dispatch_continuation_voucher_adopt(dc, ov, dc_flags);
        if (dc_flags & DISPATCH_OBJ_CONSUME_BIT) {
            dc1 = _dispatch_continuation_free_cacheonly(dc);
        } else {
```

```
            dc1 = NULL;
        }
        // 后面分析dispatch_group_async的时候会走if这个分支，但这次走的是else分支
        if (unlikely(dc_flags & DISPATCH_OBJ_GROUP_BIT)) {
            _dispatch_continuation_with_group_invoke(dc);
        } else {
            // 这次走这里，直接执行block函数
            _dispatch_client_callout(dc->dc_ctxt, dc->dc_func);
            _dispatch_introspection_queue_item_complete(dou);
        }
        if (unlikely(dc1)) {
            _dispatch_continuation_free_to_cache_limit(dc1);
        }
    });
    _dispatch_perfmon_workitem_inc();
}
```

至此，任务怎么被调度执行的已经看明白了。start_wqthread是汇编写的，直接和内核交互。虽然我们明确了使用了异步的任务被执行的调用顺序，但是想必还是有这样的疑问_dispatch_worker_thread3是怎么跟内核扯上关系的。为什么调用的是_dispatch_worker_thread3，而不是_dispatch_worker_thread或者_dispatch_worker_thread4呢?

在此之前需要说的是，在GCD中一共有2个线程池管理着任务，一个是主线程池，另一个就是除了主线程任务的线程池。主线程池由序号1的队列管理，其他有序号2的队列进行管理。加上runloop运行的runloop队列，一共就有16个队列。

```
序号   标签
1    com.apple.main-thread
2    com.apple.libdispatch-manager
3    com.apple.root.libdispatch-manager
4    com.apple.root.maintenance-qos
5    com.apple.root.maintenance-qos.overcommit
6    com.apple.root.background-qos
7    com.apple.root.background-qos.overcommit
8    com.apple.root.utility-qos
9    com.apple.root.utility-qos.overcommit
10   com.apple.root.default-qos
11   com.apple.root.default-qos.overcommit
12   com.apple.root.user-initiated-qos
13   com.apple.root.user-initiated-qos.overcommit
14   com.apple.root.user-interactive-qos
15   com.apple.root.user-interactive-qos.overcommit
```

看图的话，就如下图线程池图

有那么多root队列，所以application启动的时候就会初始化这些root队列的_dispatch_root_queues_init函数。

```
void
_dispatch_root_queues_init(void)
{
    static dispatch_once_t _dispatch_root_queues_pred;
    dispatch_once_f(&_dispatch_root_queues_pred, NULL,
            _dispatch_root_queues_init_once);
}

static void
_dispatch_root_queues_init_once(void *context DISPATCH_UNUSED)
{
    int wq_supported;
    _dispatch_fork_becomes_unsafe();
    if (!_dispatch_root_queues_init_workq(&wq_supported)) {
        size_t i;
        for (i = 0; i < DISPATCH_ROOT_QUEUE_COUNT; i++) {
            bool overcommit = true;
            _dispatch_root_queue_init_pthread_pool(
                    &_dispatch_root_queue_contexts[i], 0, overcommit);
        }
        DISPATCH_INTERNAL_CRASH((errno << 16) | wq_supported,
                "Root queue initialization failed");
    }
}

static inline bool
_dispatch_root_queues_init_workq(int *wq_supported)
{
    int r; (void)r;
    bool result = false;
    *wq_supported = 0;
    bool disable_wq = false; (void)disable_wq;
    bool disable_qos = false;
    bool disable_kevent_wq = false;
    if (!disable_wq && !disable_qos) {
        *wq_supported = _pthread_workqueue_supported();
        if (!disable_kevent_wq && (*wq_supported & WORKQ_FEATURE_KEVENT)) {
            r = _pthread_workqueue_init_with_kevent(_dispatch_worker_thread3,
                    (pthread_workqueue_function_kevent_t)
                    _dispatch_kevent_worker_thread,
                    offsetof(struct dispatch_queue_s, dq_serialnum), 0);
            result = !r;
```

```
        }
    }
    return result;
}
```

来到这里，已经看到_pthread_workqueue_init_with_kevent函数就是绑定了_dispatch_worker_thread3函数去做一些GCD的线程任务，看到源代码_pthread_workqueue_init_with_kevent做了些什么。

```
int
_pthread_workqueue_init_with_kevent(pthread_workqueue_function2_t queue_func,
        pthread_workqueue_function_kevent_t kevent_func,
        int offset, int flags)
{
    return _pthread_workqueue_init_with_workloop(queue_func, kevent_func, NUL
L, offset, flags);
}

int
_pthread_workqueue_init_with_workloop(pthread_workqueue_function2_t queue_fun
c,
        pthread_workqueue_function_kevent_t kevent_func,
        pthread_workqueue_function_workloop_t workloop_func,
        int offset, int flags)
{
    if (flags != 0) {
        return ENOTSUP;
    }

    __workq_newapi = true;
    __libdispatch_offset = offset;

    int rv = pthread_workqueue_setdispatch_with_workloop_np(queue_func, keven
t_func, workloop_func);
    return rv;
}

static int
pthread_workqueue_setdispatch_with_workloop_np(pthread_workqueue_function2_t
queue_func,
        pthread_workqueue_function_kevent_t kevent_func,
        pthread_workqueue_function_workloop_t workloop_func)
{
    int res = EBUSY;
    if (__libdispatch_workerfunction == NULL) {
```

```
        // Check whether the kernel supports new SPIs
        res = __workq_kernreturn(WQOPS_QUEUE_NEWSPISUPP, NULL, __libdispatch_
offset, kevent_func != NULL ? 0x01 : 0x00);
        if (res == -1){
            res = ENOTSUP;
        } else {
            __libdispatch_workerfunction = queue_func;
            __libdispatch_keventfunction = kevent_func;
            __libdispatch_workloopfunction = workloop_func;

            // Prepare the kernel for workq action
            (void)__workq_open();
            if (__is_threaded == 0) {
                __is_threaded = 1;
            }
        }
    }
    return res;
}
```

我们看到了__libdispatch_workerfunction = queue_func;指定了队列工作函数。然后我们往回看之前说的我们制造了一个人为crash，追溯栈里看到_pthread_wqthread这个函数。看下这个函数怎么启用_dispatch_worker_thread3的

```
// 实际代码很多，这里我精简了下，拿到了__libdispatch_workerfunction对应的_dispatch_w
orker_thread3，然后直接执行。
void
_pthread_wqthread(pthread_t self, mach_port_t kport, void *stacklowaddr, void
 *keventlist, int flags, int nkevents)
{
    pthread_workqueue_function_t func = (pthread_workqueue_function_t)__libdi
spatch_workerfunction;
    int options = overcommit ? WORKQ_ADDTHREADS_OPTION_OVERCOMMIT : 0;
    // 执行函数
    (*func)(thread_class, options, NULL);
    __workq_kernreturn(WQOPS_THREAD_RETURN, NULL, 0, 0);
    _pthread_exit(self, NULL);
}
```

0x05 dispatch_group_async

同样从入口看起

```
void dispatch_group_async(dispatch_group_t dg, dispatch_queue_t dq,
        dispatch_block_t db)
```

```
{
    dispatch_continuation_t dc = _dispatch_continuation_alloc();
    uintptr_t dc_flags = DISPATCH_OBJ_CONSUME_BIT | DISPATCH_OBJ_GROUP_BIT;

    _dispatch_continuation_init(dc, dq, db, 0, 0, dc_flags);
    _dispatch_continuation_group_async(dg, dq, dc);
}
```

同样是_dispatch_continuation_init函数，这里跟dispatch_async那里一毛一样，忘记了的话，往回看。我们接着往下看，_dispatch_continuation_group_async函数

```
static inline void
_dispatch_continuation_group_async(dispatch_group_t dg, dispatch_queue_t dq,
        dispatch_continuation_t dc)
{
    dispatch_group_enter(dg);
    dc->dc_data = dg;
    _dispatch_continuation_async(dq, dc);
}
```

我们发现，其实dispatch_group_async内部也是加了dispatch_group_enter函数。dispatch_group_async怎么初始化我们至此已经说明完毕。

后面取出执行block逻辑跟dispatch_async略微不同，前面部分不做多说，调用顺序跟dispatch_async是一样的，唯一不同在于_dispatch_continuation_invoke_inline这个函数。

```
static inline void _dispatch_continuation_invoke_inline(dispatch_object_t dou
, voucher_t ov,
        dispatch_invoke_flags_t flags)
{
    dispatch_continuation_t dc = dou._dc, dc1;
    dispatch_invoke_with_autoreleasepool(flags, {
        uintptr_t dc_flags = dc->dc_flags;

        _dispatch_continuation_voucher_adopt(dc, ov, dc_flags);
        if (dc_flags & DISPATCH_OBJ_CONSUME_BIT) {
            dc1 = _dispatch_continuation_free_cacheonly(dc);
        } else {
            dc1 = NULL;
        }
        // 这次走if语句
        if (unlikely(dc_flags & DISPATCH_OBJ_GROUP_BIT)) {
            _dispatch_continuation_with_group_invoke(dc);
        } else {
            _dispatch_client_callout(dc->dc_ctxt, dc->dc_func);
```

```
            _dispatch_introspection_queue_item_complete(dou);
        }
        if (unlikely(dc1)) {
            _dispatch_continuation_free_to_cache_limit(dc1);
        }
    });
    _dispatch_perfmon_workitem_inc();
}

static inline void
_dispatch_continuation_with_group_invoke(dispatch_continuation_t dc)
{
    struct dispatch_object_s *dou = dc->dc_data;
    unsigned long type = dx_type(dou);
    if (type == DISPATCH_GROUP_TYPE) {
        // 执行任务
        _dispatch_client_callout(dc->dc_ctxt, dc->dc_func);
        _dispatch_introspection_queue_item_complete(dou);
        // 调用dispatch_group_leave
        dispatch_group_leave((dispatch_group_t)dou);
    } else {
        DISPATCH_INTERNAL_CRASH(dx_type(dou), "Unexpected object type");
    }
}
```

我们有必要看下dispatch_group_enter和dispatch_group_leave函数。

```
// dispatch_group_enter里面没啥说的，也就dg->dg_value值加1
void dispatch_group_enter(dispatch_group_t dg)
{
    long value = os_atomic_inc_orig2o(dg, dg_value, acquire);
    if (slowpath((unsigned long)value >= (unsigned long)LONG_MAX)) {
        DISPATCH_CLIENT_CRASH(value,
                "Too many nested calls to dispatch_group_enter()");
    }
    if (value == 0) {
        _dispatch_retain(dg);
    }
}

void dispatch_group_leave(dispatch_group_t dg)
{
    long value = os_atomic_dec2o(dg, dg_value, release);
    // 如果没有等待者，则调用_dispatch_group_wake函数
    if (slowpath(value == 0)) {
        return (void)_dispatch_group_wake(dg, true);
```

```
    }
    // 小于0就会crash，所以dispatch_group_enter和dispatch_group_leave必须匹配，不然就crash了。
    if (slowpath(value < 0)) {
        DISPATCH_CLIENT_CRASH(value,
                "Unbalanced call to dispatch_group_leave()");
    }
}

static long _dispatch_group_wake(dispatch_group_t dg, bool needs_release)
{
    dispatch_continuation_t next, head, tail = NULL;
    long rval;


    head = os_atomic_xchg2o(dg, dg_notify_head, NULL, relaxed);
    if (head) {
        tail = os_atomic_xchg2o(dg, dg_notify_tail, NULL, release);
    }
    // dg->dg_waiters赋值为0，并返回dg->dg_waiters之前的值
    rval = (long)os_atomic_xchg2o(dg, dg_waiters, 0, relaxed);
    // 如果之前还有等待者
    if (rval) {
        // 创建信号量
        _dispatch_sema4_create(&dg->dg_sema, _DSEMA4_POLICY_FIFO);
        // 发出信号
        _dispatch_sema4_signal(&dg->dg_sema, rval);
    }
    uint16_t refs = needs_release ? 1 : 0;
    // dispatch_group里是否有任务等待执行，有的话加入。
    // 比如dispatch_group_notify的任务就在此时被唤醒
    if (head) {

        do {
            next = os_mpsc_pop_snapshot_head(head, tail, do_next);
            dispatch_queue_t dsn_queue = (dispatch_queue_t)head->dc_data;
            _dispatch_continuation_async(dsn_queue, head);
            _dispatch_release(dsn_queue);
        } while ((head = next));
        refs++;
    }
    if (refs) _dispatch_release_n(dg, refs);
    return 0;
}
```

0x06 dispatch_once

还是从入口函数开始看

```
// 我们调用dispatch_once的入口
void dispatch_once(dispatch_once_t *val, dispatch_block_t block)
{
// 内部又调用了dispatch_once_f函数
dispatch_once_f(val, block, _dispatch_Block_invoke(block));
}

DISPATCH_NOINLINE
void dispatch_once_f(dispatch_once_t *val, void *ctxt, dispatch_function_t func) {
return dispatch_once_f_slow(val, ctxt, func);
}

DISPATCH_ONCE_SLOW_INLINE
static void
dispatch_once_f_slow(dispatch_once_t *val, void *ctxt, dispatch_function_t func)
{
// _dispatch_once_waiter_t格式：
// typedef struct _dispatch_once_waiter_s {
// volatile struct _dispatch_once_waiter_s *volatile dow_next;
// dispatch_thread_event_s dow_event;
// mach_port_t dow_thread;
// } *_dispatch_once_waiter_t;

// volatile：告诉编译器不要对此指针进行代码优化，因为这个指针指向的值可能会被其他线程改变
_dispatch_once_waiter_t volatile *vval = (_dispatch_once_waiter_t*)val;
struct _dispatch_once_waiter_s dow = { };
_dispatch_once_waiter_t tail = &dow, next, tmp;
dispatch_thread_event_t event;

// 第一次执行时，*vval为0，此时第一个参数vval和第二个参数NULL比较是相等的，返回true，然后把tail赋值给第一个参数的值。如果这时候同时有别的线程也进来，此时vval的值不是0了，所以会来到else分支。
if (os_atomic_cmpxchg(vval, NULL, tail, acquire)) {
    // 获取当前线程
    dow.dow_thread = _dispatch_tid_self();
    // 调用block函数，一般就是我们在外面做的初始化工作
    _dispatch_client_callout(ctxt, func);

    // 内部将DLOCK_ONCE_DONE赋值给val，将当前标记为已完成，返回之前的引用值。前面说过了
```

，把tail赋值给val了，但这只是没有别的线程进来走到下面else分支，如果有别的线程进来next就是别的值了，如果没有别的信号量在等待，工作就到此结束了。

```
    next = (_dispatch_once_waiter_t)_dispatch_once_xchg_done(val);
    // 如果没有别的线程进来过处于等待，这里就会结束。如果有，则遍历每一个等待的信号量，然后一个个唤醒它们
    while (next != tail) {
        // 内部用到了thread_switch，避免优先级反转。把next->dow_next返回
        tmp = (_dispatch_once_waiter_t)_dispatch_wait_until(next->dow_next);
        event = &next->dow_event;
        next = tmp;
        // 唤醒信号量
        _dispatch_thread_event_signal(event);
    }
} else {
    // 内部就是_dispatch_sema4_init函数，也就是初始化一个信号链表
    _dispatch_thread_event_init(&dow.dow_event);
    // next指向新的原子
    next = *vval;
    // 不断循环等待
    for (;;) {
        // 前面说过第一次进来后进入if分支，后面再次进来，会来到这里，但是之前if里面被标志为DISPATCH_ONCE_DONE了，所以结束。
        if (next == DISPATCH_ONCE_DONE) {
            break;
        }
        // 当第一次初始化的时候，同时有别的线程也进来，这是第一个线程已经占据了if分支，但其他线程也是第一进来，所以状态并不是DISPATCH_ONCE_DONE，所以就来到了这里
        // 比较vval和next是否一样，其他线程第一次来这里肯定是相等的
        if (os_atomic_cmpxchgv(vval, next, tail, &next, release)) {
            dow.dow_thread = next->dow_thread;
            dow.dow_next = next;
            if (dow.dow_thread) {
                pthread_priority_t pp = _dispatch_get_priority();
                _dispatch_thread_override_start(dow.dow_thread, pp, val);
            }
            // 等待唤醒，唤醒后就做收尾操作
            _dispatch_thread_event_wait(&dow.dow_event);
            if (dow.dow_thread) {

                _dispatch_thread_override_end(dow.dow_thread, val);
            }
            break;
        }
    }
    // 销毁信号量
    _dispatch_thread_event_destroy(&dow.dow_event);
```

```
}
```

}

那么，回到上篇提到使用dispatch_once死锁的问题，如果使用不当会造成什么后果？回顾下上篇的实验代码

- (void)viewDidLoad {
  [super viewDidLoad];

  [self once];
  }

- (void)once {
  static dispatch_once_t onceToken;
  dispatch_once(&onceToken, {
  [self otherOnce];
  });
  NSLog(@"遇到第一只熊猫宝宝...");
  }

- (void)otherOnce {
  static dispatch_once_t onceToken;
  dispatch_once(&onceToken, {
  [self once];
  });
  NSLog(@"遇到第二只熊猫宝宝...");
  }

示例中我们可以看到once方法需要等待otherOnce方法的完成，而otherOnce又调用了once，根据前面的源码，otherOnce调用once方法会走到else分支，在这个分支等待之前一个信号量发出唤醒指令，但是once方法里面又依赖otherOnce方法的完成，由于处于一个线程，所以就卡住了。

0x06 dispatch_group_create & dispatch_semaphore_create

为什么两个一起看，其实dispatch_group也是通过dispatch_semaphore控制的，看下dispatch_group_create源代码：

```
dispatch_group_t
dispatch_group_create(void)
{
    return _dispatch_group_create_with_count(0);
}

static inline dispatch_group_t
_dispatch_group_create_with_count(long count)
{
    dispatch_group_t dg = (dispatch_group_t)_dispatch_object_alloc(
        DISPATCH_VTABLE(group), sizeof(struct dispatch_group_s));
 _dispatch_semaphore_class_init(count, dg); // 初始化信号量
    if (count) {
        os_atomic_store2o(dg, do_ref_cnt, 1, relaxed);
    }
    return dg;
}
```

同样的，看下dispatch_semaphore_create源代码，是不是一股熟悉的配方：

```
    dispatch_semaphore_t
    dispatch_semaphore_create(long value)
    {
        dispatch_semaphore_t dsema;

        if (value < 0) {
            return DISPATCH_BAD_INPUT;
        }

        dsema = (dispatch_semaphore_t)_dispatch_object_alloc(
            DISPATCH_VTABLE(semaphore), sizeof(struct dispatch_semaphore_s));
        _dispatch_semaphore_class_init(value, dsema); // 同样的初始化信号量
        dsema->dsema_orig = value;
        return dsema;
}

0x07 dispatch_group_wait & dispatch_semaphore_wait
```

再看下dispatch_group_wait的代码，其内部是调用的_dispatch_group_wait_slow函数：

```
static long _dispatch_group_wait_slow(dispatch_group_t dg, dispatch_time_t ti
meout)
{
    long value;
    int orig_waiters;
```

```
    value = os_atomic_load2o(dg, dg_value, ordered);
    if (value == 0) {
        return _dispatch_group_wake(dg, false);
    }

    (void)os_atomic_inc2o(dg, dg_waiters, relaxed);

    value = os_atomic_load2o(dg, dg_value, ordered);
    // 如果group里没有任务
    if (value == 0) {
        _dispatch_group_wake(dg, false);

        timeout = DISPATCH_TIME_FOREVER;
    }

    _dispatch_sema4_create(&dg->dg_sema, _DSEMA4_POLICY_FIFO);
    switch (timeout) {
    default:
        if (!_dispatch_sema4_timedwait(&dg->dg_sema, timeout)) {
            break;
        }

    case DISPATCH_TIME_NOW:
        orig_waiters = dg->dg_waiters;
        while (orig_waiters) {
            if (os_atomic_cmpxchgvw2o(dg, dg_waiters, orig_waiters,
                    orig_waiters - 1, &orig_waiters, relaxed)) {
                return _DSEMA4_TIMEOUT();
            }
        }

    case DISPATCH_TIME_FOREVER:
        _dispatch_sema4_wait(&dg->dg_sema);
        break;
    }
    return 0;
}
```

对比着看dispatch_semaphore_wait源码，其内部也调用_dispatch_semaphore_wait_slow函数，可以看到逻辑基本一致：

```
static long _dispatch_semaphore_wait_slow(dispatch_semaphore_t dsema,
        dispatch_time_t timeout)
{
    long orig;
```

```
    _dispatch_sema4_create(&dsema->dsema_sema, _DSEMA4_POLICY_FIFO);
    switch (timeout) {
    default:
        if (!_dispatch_sema4_timedwait(&dsema->dsema_sema, timeout)) {
            break;
        }

    case DISPATCH_TIME_NOW:
        orig = dsema->dsema_value;
        while (orig < 0) {
            if (os_atomic_cmpxchgvw2o(dsema, dsema_value, orig, orig + 1,
                    &orig, relaxed)) {
                return _DSEMA4_TIMEOUT();
            }
        }

    case DISPATCH_TIME_FOREVER:
        _dispatch_sema4_wait(&dsema->dsema_sema);
        break;
    }
    return 0;
}
```

0x08 dispatch_group_notify

再把dispatch_group_notify看下

```
void
dispatch_group_notify(dispatch_group_t dg, dispatch_queue_t dq,
        dispatch_block_t db)
{
    dispatch_continuation_t dsn = _dispatch_continuation_alloc();
    // 之前说过了
    _dispatch_continuation_init(dsn, dq, db, 0, 0, DISPATCH_OBJ_CONSUME_BIT);
    _dispatch_group_notify(dg, dq, dsn);
}

static inline void _dispatch_group_notify(dispatch_group_t dg, dispatch_queue
_t dq,
        dispatch_continuation_t dsn)
{
    dsn->dc_data = dq;
    dsn->do_next = NULL;
    _dispatch_retain(dq);
    if (os_mpsc_push_update_tail(dg, dg_notify, dsn, do_next)) {
```

```
        _dispatch_retain(dg);
        os_atomic_store2o(dg, dg_notify_head, dsn, ordered);
        // 如果此时group里面的任务都完成了，那么就立刻唤醒
        if (os_atomic_load2o(dg, dg_value, ordered) == 0) {
            _dispatch_group_wake(dg, false);
        }
    }
}
```

在dispatch_group_async里面我们知道dispatch_group的任务在执行后会调用dispatch_group_leave。这个函数里面如果等待者没有了，就会唤醒dispatch_group。里面的任务，比如dispatch_group_notify的任务就会这时候被执行。

这里执行的调用顺序就不贴了，基本跟dispatch_async一致。

0x09 dispatch_barrier_async

可以看到，大多数实现都是大同小异，通过不同的标志位来控制。这里跟dispatch_async的不同就在于，dispatch_async直接把任务扔到root队列，而dispatch_barrier_async是把任务在到自定义的队列。

```
void
dispatch_barrier_async(dispatch_queue_t dq, dispatch_block_t work)
{
    dispatch_continuation_t dc = _dispatch_continuation_alloc();
    uintptr_t dc_flags = DISPATCH_OBJ_CONSUME_BIT | DISPATCH_OBJ_BARRIER_BIT;

    _dispatch_continuation_init(dc, dq, work, 0, 0, dc_flags);
    _dispatch_continuation_push(dq, dc);
}

void _dispatch_queue_push(dispatch_queue_t dq, dispatch_object_t dou,
        dispatch_qos_t qos)
{
    _dispatch_queue_push_inline(dq, dou, qos);
}

static inline void _dispatch_queue_push_inline(dispatch_queue_t dq, dispatch_
object_t _tail,
        dispatch_qos_t qos)
{
    struct dispatch_object_s *tail = _tail._do;
    dispatch_wakeup_flags_t flags = 0;
    bool overriding = _dispatch_queue_need_override_retain(dq, qos);
    // 加入到自己的队列
    if (unlikely(_dispatch_queue_push_update_tail(dq, tail))) {
```

```
        if (!overriding) _dispatch_retain_2(dq->_as_os_obj);
        _dispatch_queue_push_update_head(dq, tail);
        flags = DISPATCH_WAKEUP_CONSUME_2 | DISPATCH_WAKEUP_MAKE_DIRTY;
    } else if (overriding) {
        flags = DISPATCH_WAKEUP_CONSUME_2;
    } else {
        return;
    }
    // 唤醒队列
    return dx_wakeup(dq, qos, flags);
}

void _dispatch_queue_wakeup(dispatch_queue_t dq, dispatch_qos_t qos,
        dispatch_wakeup_flags_t flags)
{
    dispatch_queue_wakeup_target_t target = DISPATCH_QUEUE_WAKEUP_NONE;

    if (unlikely(flags & DISPATCH_WAKEUP_BARRIER_COMPLETE)) {
        return _dispatch_queue_barrier_complete(dq, qos, flags);
    }
    // 内部就是 tail != NULL，所以满足条件
    if (_dispatch_queue_class_probe(dq)) {
        // #define DISPATCH_QUEUE_WAKEUP_TARGET  ((dispatch_queue_wakeup_targ
et_t)1)
        target = DISPATCH_QUEUE_WAKEUP_TARGET;
    }
    return _dispatch_queue_class_wakeup(dq, qos, flags, target);
}

void _dispatch_queue_class_wakeup(dispatch_queue_t dq, dispatch_qos_t qos,
        dispatch_wakeup_flags_t flags, dispatch_queue_wakeup_target_t target)
{
    dispatch_assert(target != DISPATCH_QUEUE_WAKEUP_WAIT_FOR_EVENT);

    // 会走进去
    if (target) {
        uint64_t old_state, new_state, enqueue = DISPATCH_QUEUE_ENQUEUED;
        if (target == DISPATCH_QUEUE_WAKEUP_MGR) {
            enqueue = DISPATCH_QUEUE_ENQUEUED_ON_MGR;
        }
        qos = _dispatch_queue_override_qos(dq, qos);
        os_atomic_rmw_loop2o(dq, dq_state, old_state, new_state, release, {
            new_state = _dq_state_merge_qos(old_state, qos);
            if (likely(!_dq_state_is_suspended(old_state) &&
                    !_dq_state_is_enqueued(old_state) &&
                    (!_dq_state_drain_locked(old_state) ||
```

```
                    (enqueue != DISPATCH_QUEUE_ENQUEUED_ON_MGR &&
                    _dq_state_is_base_wlh(old_state))))) {
                new_state |= enqueue;
            }
            if (flags & DISPATCH_WAKEUP_MAKE_DIRTY) {
                new_state |= DISPATCH_QUEUE_DIRTY;
            } else if (new_state == old_state) {
                os_atomic_rmw_loop_give_up(goto done);
            }
        });

        if (likely((old_state ^ new_state) & enqueue)) {
            dispatch_queue_t tq;
            if (target == DISPATCH_QUEUE_WAKEUP_TARGET) {
                os_atomic_thread_fence(dependency);
                tq = os_atomic_load_with_dependency_on2o(dq, do_targetq,
                        (long)new_state);
            } else {
                tq = target;
            }
            dispatch_assert(_dq_state_is_enqueued(new_state));
            // 把队列装入到root队列中,内部调用的_dispatch_root_queue_push函数
            return _dispatch_queue_push_queue(tq, dq, new_state);
        }
    }
done:
    if (likely(flags & DISPATCH_WAKEUP_CONSUME_2)) {
        return _dispatch_release_2_tailcall(dq);
    }
}

// _dispatch_root_queue_push函数在dispatch_async已经贴过代码，直接看_dispatch_roo
t_queue_push_override函数
static void _dispatch_root_queue_push_override(dispatch_queue_t orig_rq,
        dispatch_object_t dou, dispatch_qos_t qos)
{
    bool overcommit = orig_rq->dq_priority & DISPATCH_PRIORITY_FLAG_OVERCOMMI
T;
    dispatch_queue_t rq = _dispatch_get_root_queue(qos, overcommit);
    dispatch_continuation_t dc = dou._dc;
    // 因为barrier是直接推进自己的队列，所以这里不会走if语句，具体注释可以看dispatch_as
ync那里
    if (_dispatch_object_is_redirection(dc)) {
        dc->dc_func = (void *)orig_rq;
    } else {
        dc = _dispatch_continuation_alloc();
```

```
        // 指定do_vtable，所以取出来执行的时候调用的是_dispatch_queue_override_invo
ke函数
        dc->do_vtable = DC_VTABLE(OVERRIDE_OWNING);
        _dispatch_trace_continuation_push(orig_rq, dou);
        dc->dc_ctxt = dc;
        dc->dc_other = orig_rq;
        dc->dc_data = dou._do;
        dc->dc_priority = DISPATCH_NO_PRIORITY;
        dc->dc_voucher = DISPATCH_NO_VOUCHER;
    }
    _dispatch_root_queue_push_inline(rq, dc, dc, 1);
}
```

// 后面也省略

同样我们人为制造一个闪退，看下被调用顺序

```
6   libdispatch.dylib                   0x0000000105b952f7 _dispatch_call_blo
ck_and_release + 12
7   libdispatch.dylib                   0x0000000105b9633d _dispatch_client_c
allout + 8
8   libdispatch.dylib                   0x0000000105ba40a5 _dispatch_queue_co
ncurrent_drain + 1492
9   libdispatch.dylib                   0x0000000105b9f1fb _dispatch_queue_in
voke + 353
10  libdispatch.dylib                   0x0000000105b9af7c _dispatch_queue_ov
erride_invoke + 733
11  libdispatch.dylib                   0x0000000105ba2102 _dispatch_root_que
ue_drain + 772
12  libdispatch.dylib                   0x0000000105ba1da0 _dispatch_worker_t
hread3 + 132
13  libsystem_pthread.dylib             0x000000010605d5a2 _pthread_wqthread
+ 1299
14  libsystem_pthread.dylib             0x000000010605d07d start_wqthread + 1
3
```

一样从_dispatch_root_queue_drain开始看

```
 static void _dispatch_root_queue_drain(dispatch_queue_t dq, pthread_priority
_t pp)
{
    _dispatch_queue_set_current(dq);
    dispatch_priority_t pri = dq->dq_priority;
    if (!pri) pri = _dispatch_priority_from_pp(pp);
    dispatch_priority_t old_dbp = _dispatch_set_basepri(pri);
```

```
    _dispatch_adopt_wlh_anon();

    struct dispatch_object_s *item;
    bool reset = false;
    dispatch_invoke_context_s dic = { };
    dispatch_invoke_flags_t flags = DISPATCH_INVOKE_WORKER_DRAIN |
            DISPATCH_INVOKE_REDIRECTING_DRAIN;
    _dispatch_queue_drain_init_narrowing_check_deadline(&dic, pri);
    _dispatch_perfmon_start();
    // rootqueue可以跟一个dispatch_queue_t也可以跟一个dispatch_continuation_t
    // 所以这里item取出来的是dispatch_queue_t
    while ((item = fastpath(_dispatch_root_queue_drain_one(dq)))) {
        if (reset) _dispatch_wqthread_override_reset();
        _dispatch_continuation_pop_inline(item, &dic, flags, dq);
        // 重置当前线程的优先级，会跟内核交互
        reset = _dispatch_reset_basepri_override();
        if (unlikely(_dispatch_queue_drain_should_narrow(&dic))) {
            break;
        }
    }

    // overcommit or not. worker thread
    if (pri & _PTHREAD_PRIORITY_OVERCOMMIT_FLAG) {
        _dispatch_perfmon_end(perfmon_thread_worker_oc);
    } else {
        _dispatch_perfmon_end(perfmon_thread_worker_non_oc);
    }

    _dispatch_reset_wlh();
    _dispatch_reset_basepri(old_dbp);
    _dispatch_reset_basepri_override();
    _dispatch_queue_set_current(NULL);
}

static inline void _dispatch_continuation_pop_inline(dispatch_object_t dou,
        dispatch_invoke_context_t dic, dispatch_invoke_flags_t flags,
        dispatch_queue_t dq)
{
    dispatch_pthread_root_queue_observer_hooks_t observer_hooks =
            _dispatch_get_pthread_root_queue_observer_hooks();
    if (observer_hooks) observer_hooks->queue_will_execute(dq);
    _dispatch_trace_continuation_pop(dq, dou);
    flags &= _DISPATCH_INVOKE_PROPAGATE_MASK;
    // 调用_dispatch_queue_override_invoke函数
    // 这里其实很好理解，从root队列拿出来的有可能是一个队列，也可能就是一个任务，所以如果是队列，就调用队列的执行函数
```

```
    // 所以为什么官方文档说，不是自定义队列使用barrier无效，因为不是自定义队列，这里就直接
走_dispatch_continuation_invoke_inline函数，调用函数实现了，也就是dispatch_barrier
_async类似于dispatch_async了。
    if (_dispatch_object_has_vtable(dou)) {
        dx_invoke(dou._do, dic, flags);
    } else {
        _dispatch_continuation_invoke_inline(dou, DISPATCH_NO_VOUCHER, flags)
;
    }
    if (observer_hooks) observer_hooks->queue_did_execute(dq);
}

void _dispatch_queue_override_invoke(dispatch_continuation_t dc,
        dispatch_invoke_context_t dic, dispatch_invoke_flags_t flags)
{
    dispatch_queue_t old_rq = _dispatch_queue_get_current();
    dispatch_queue_t assumed_rq = dc->dc_other;
    dispatch_priority_t old_dp;
    voucher_t ov = DISPATCH_NO_VOUCHER;
    dispatch_object_t dou;

    dou._do = dc->dc_data;
    // 将自定义queue激活，其root队列挂起。将rootqueue保存到old_dq变量
    // 所以这也就是为什么，barrier的任务可以提前执行，后面的任务会被阻塞
    //  static inline dispatch_priority_t
    //_dispatch_root_queue_identity_assume(dispatch_queue_t assumed_rq)
    //{
    //      dispatch_priority_t old_dbp = _dispatch_get_basepri();
    //      dispatch_assert(dx_hastypeflag(assumed_rq, QUEUE_ROOT));
    //      _dispatch_reset_basepri(assumed_rq->dq_priority);
    //      _dispatch_queue_set_current(assumed_rq);
    //      return old_dbp;
    //}
    old_dp = _dispatch_root_queue_identity_assume(assumed_rq);
    if (dc_type(dc) == DISPATCH_CONTINUATION_TYPE(OVERRIDE_STEALING)) {
        flags |= DISPATCH_INVOKE_STEALING;
    } else {
        _dispatch_trace_continuation_pop(assumed_rq, dou._do);
    }
    _dispatch_continuation_pop_forwarded(dc, ov, DISPATCH_OBJ_CONSUME_BIT, {
        // 来到if分支，调用_dispatch_queue_invoke函数
        if (_dispatch_object_has_vtable(dou._do)) {
            dx_invoke(dou._do, dic, flags);
        } else {
            _dispatch_continuation_invoke_inline(dou, ov, flags);
        }
```

```
    });
    // 重新激活root队列
    _dispatch_reset_basepri(old_dp);
    _dispatch_queue_set_current(old_rq);
}

void _dispatch_queue_invoke(dispatch_queue_t dq, dispatch_invoke_context_t di
c,
        dispatch_invoke_flags_t flags)
{
    _dispatch_queue_class_invoke(dq, dic, flags, 0, dispatch_queue_invoke2);
}

static inline void _dispatch_queue_class_invoke(dispatch_object_t dou,
        dispatch_invoke_context_t dic, dispatch_invoke_flags_t flags,
        dispatch_invoke_flags_t const_restrict_flags,
        _dispatch_queue_class_invoke_handler_t invoke)
{
    dispatch_queue_t dq = dou._dq;
    dispatch_queue_wakeup_target_t tq = DISPATCH_QUEUE_WAKEUP_NONE;
    bool owning = !(flags & DISPATCH_INVOKE_STEALING);
    uint64_t owned = 0;

    if (!(flags & (DISPATCH_INVOKE_STEALING | DISPATCH_INVOKE_WLH))) {
        dq->do_next = DISPATCH_OBJECT_LISTLESS;
    }
    flags |= const_restrict_flags;
    if (likely(flags & DISPATCH_INVOKE_WLH)) {
        owned = DISPATCH_QUEUE_SERIAL_DRAIN_OWNED | DISPATCH_QUEUE_ENQUEUED;
    } else {
        owned = _dispatch_queue_drain_try_lock(dq, flags);
    }
    if (likely(owned)) {
        dispatch_priority_t old_dbp;
        if (!(flags & DISPATCH_INVOKE_MANAGER_DRAIN)) {
            old_dbp = _dispatch_set_basepri(dq->dq_priority);
        } else {
            old_dbp = 0;
        }

        flags = _dispatch_queue_merge_autorelease_frequency(dq, flags);

attempt_running_slow_head:
        // 执行dispatch_queue_invoke2函数
        // 也就是执行自定义队列里面的任务
        tq = invoke(dq, dic, flags, &owned);
```

```
        dispatch_assert(tq != DISPATCH_QUEUE_WAKEUP_TARGET);
        if (unlikely(tq != DISPATCH_QUEUE_WAKEUP_NONE &&
                tq != DISPATCH_QUEUE_WAKEUP_WAIT_FOR_EVENT)) {
        } else if (!_dispatch_queue_drain_try_unlock(dq, owned,
                tq == DISPATCH_QUEUE_WAKEUP_NONE)) {
            tq = _dispatch_queue_get_current();
            if (dx_hastypeflag(tq, QUEUE_ROOT) || !owning) {
                goto attempt_running_slow_head;
            }
            DISPATCH_COMPILER_CAN_ASSUME(tq != DISPATCH_QUEUE_WAKEUP_NONE);
        } else {
            owned = 0;
            tq = NULL;
        }
        if (!(flags & DISPATCH_INVOKE_MANAGER_DRAIN)) {
            _dispatch_reset_basepri(old_dbp);
        }
    }
    if (likely(owning)) {
        _dispatch_introspection_queue_item_complete(dq);
    }

    if (tq) {
        if (const_restrict_flags & DISPATCH_INVOKE_DISALLOW_SYNC_WAITERS) {
            dispatch_assert(dic->dic_deferred == NULL);
        } else if (dic->dic_deferred) {
            return _dispatch_queue_drain_sync_waiter(dq, dic,
                    flags, owned);
        }

        uint64_t old_state, new_state, enqueued = DISPATCH_QUEUE_ENQUEUED;
        if (tq == DISPATCH_QUEUE_WAKEUP_MGR) {
            enqueued = DISPATCH_QUEUE_ENQUEUED_ON_MGR;
        }
        os_atomic_rmw_loop2o(dq, dq_state, old_state, new_state, release, {
            new_state  = old_state - owned;
            new_state &= ~DISPATCH_QUEUE_DRAIN_UNLOCK_MASK;
            new_state |= DISPATCH_QUEUE_DIRTY;
            if (_dq_state_is_suspended(new_state)) {
                new_state |= DLOCK_OWNER_MASK;
            } else if (_dq_state_is_runnable(new_state) &&
                    !_dq_state_is_enqueued(new_state)) {
                // drain was not interupted for suspension
                // we will reenqueue right away, just put ENQUEUED back
                new_state |= enqueued;
            }
```

```
        });
        old_state -= owned;
        if (_dq_state_received_override(old_state)) {
            // Ensure that the root queue sees that this thread was overridde
n.
            _dispatch_set_basepri_override_qos(_dq_state_max_qos(new_state));
        }
        if ((old_state ^ new_state) & enqueued) {
            dispatch_assert(_dq_state_is_enqueued(new_state));
            return _dispatch_queue_push_queue(tq, dq, new_state);
        }
    }

    _dispatch_release_2_tailcall(dq);
}

static inline dispatch_queue_wakeup_target_t dispatch_queue_invoke2(dispatch_
queue_t dq, dispatch_invoke_context_t dic,
        dispatch_invoke_flags_t flags, uint64_t *owned)
{
    dispatch_queue_t otq = dq->do_targetq;
    dispatch_queue_t cq = _dispatch_queue_get_current();

    if (slowpath(cq != otq)) {
        return otq;
    }
    if (dq->dq_width == 1) {
        return _dispatch_queue_serial_drain(dq, dic, flags, owned);
    }
    return _dispatch_queue_concurrent_drain(dq, dic, flags, owned);
}

static dispatch_queue_wakeup_target_t _dispatch_queue_concurrent_drain(dispat
ch_queue_t dq,
        dispatch_invoke_context_t dic, dispatch_invoke_flags_t flags,
        uint64_t *owned)
{
    return _dispatch_queue_drain(dq, dic, flags, owned, false);
}

static dispatch_queue_wakeup_target_t
_dispatch_queue_drain(dispatch_queue_t dq, dispatch_invoke_context_t dic,
        dispatch_invoke_flags_t flags, uint64_t *owned_ptr, bool serial_drain
)
{
    dispatch_queue_t orig_tq = dq->do_targetq;
```

```
    dispatch_thread_frame_s dtf;
    struct dispatch_object_s *dc = NULL, *next_dc;
    uint64_t dq_state, owned = *owned_ptr;

    if (unlikely(!dq->dq_items_tail)) return NULL;

    _dispatch_thread_frame_push(&dtf, dq);
    if (serial_drain || _dq_state_is_in_barrier(owned)) {
        // we really own `IN_BARRIER + dq->dq_width * WIDTH_INTERVAL`
        // but width can change while draining barrier work items, so we only
        // convert to `dq->dq_width * WIDTH_INTERVAL` when we drop `IN_BARRIE
R`
        owned = DISPATCH_QUEUE_IN_BARRIER;
    } else {
        owned &= DISPATCH_QUEUE_WIDTH_MASK;
    }

    dc = _dispatch_queue_head(dq);
    goto first_iteration;

    // 循环执行自定义里面的任务，一个接一个执行，不能并行执行。
    for (;;) {
        dc = next_dc;
        if (unlikely(dic->dic_deferred)) {
            goto out_with_deferred_compute_owned;
        }
        if (unlikely(_dispatch_needs_to_return_to_kernel())) {
            _dispatch_return_to_kernel();
        }
        if (unlikely(!dc)) {
            if (!dq->dq_items_tail) {
                break;
            }
            dc = _dispatch_queue_head(dq);
        }
        if (unlikely(serial_drain != (dq->dq_width == 1))) {
            break;
        }
        if (unlikely(_dispatch_queue_drain_should_narrow(dic))) {
            break;
        }

first_iteration:
        dq_state = os_atomic_load(&dq->dq_state, relaxed);
        if (unlikely(_dq_state_is_suspended(dq_state))) {
            break;
```

```
        }
        if (unlikely(orig_tq != dq->do_targetq)) {
            break;
        }

        if (serial_drain || _dispatch_object_is_barrier(dc)) {
            if (!serial_drain && owned != DISPATCH_QUEUE_IN_BARRIER) {
                if (!_dispatch_queue_try_upgrade_full_width(dq, owned)) {
                    goto out_with_no_width;
                }
                owned = DISPATCH_QUEUE_IN_BARRIER;
            }
            next_dc = _dispatch_queue_next(dq, dc);
            if (_dispatch_object_is_sync_waiter(dc)) {
                owned = 0;
                dic->dic_deferred = dc;
                goto out_with_deferred;
            }
        } else {
            if (owned == DISPATCH_QUEUE_IN_BARRIER) {
                os_atomic_xor2o(dq, dq_state, owned, release);
                owned = dq->dq_width * DISPATCH_QUEUE_WIDTH_INTERVAL;
            } else if (unlikely(owned == 0)) {
                if (_dispatch_object_is_sync_waiter(dc)) {
                    // sync "readers" don't observe the limit
                    _dispatch_queue_reserve_sync_width(dq);
                } else if (!_dispatch_queue_try_acquire_async(dq)) {
                    goto out_with_no_width;
                }
                owned = DISPATCH_QUEUE_WIDTH_INTERVAL;
            }

            next_dc = _dispatch_queue_next(dq, dc);
            if (_dispatch_object_is_sync_waiter(dc)) {
                owned -= DISPATCH_QUEUE_WIDTH_INTERVAL;
                _dispatch_sync_waiter_redirect_or_wake(dq,
                        DISPATCH_SYNC_WAITER_NO_UNLOCK, dc);
                continue;
            }

            if (flags & DISPATCH_INVOKE_REDIRECTING_DRAIN) {
                owned -= DISPATCH_QUEUE_WIDTH_INTERVAL;
                _dispatch_continuation_redirect(dq, dc);
                continue;
            }
        }
```

```
        // 执行这个函数
        _dispatch_continuation_pop_inline(dc, dic, flags, dq);
    }

    if (owned == DISPATCH_QUEUE_IN_BARRIER) {
        // if we're IN_BARRIER we really own the full width too
        owned += dq->dq_width * DISPATCH_QUEUE_WIDTH_INTERVAL;
    }
    if (dc) {
        owned = _dispatch_queue_adjust_owned(dq, owned, dc);
    }
    *owned_ptr &= DISPATCH_QUEUE_ENQUEUED | DISPATCH_QUEUE_ENQUEUED_ON_MGR;
    *owned_ptr |= owned;
    _dispatch_thread_frame_pop(&dtf);
    return dc ? dq->do_targetq : NULL;

out_with_no_width:
    *owned_ptr &= DISPATCH_QUEUE_ENQUEUED | DISPATCH_QUEUE_ENQUEUED_ON_MGR;
    _dispatch_thread_frame_pop(&dtf);
    return DISPATCH_QUEUE_WAKEUP_WAIT_FOR_EVENT;

out_with_deferred_compute_owned:
    if (serial_drain) {
        owned = DISPATCH_QUEUE_IN_BARRIER + DISPATCH_QUEUE_WIDTH_INTERVAL;
    } else {
        if (owned == DISPATCH_QUEUE_IN_BARRIER) {
            // if we're IN_BARRIER we really own the full width too
            owned += dq->dq_width * DISPATCH_QUEUE_WIDTH_INTERVAL;
        }
        if (dc) {
            owned = _dispatch_queue_adjust_owned(dq, owned, dc);
        }
    }
out_with_deferred:
    *owned_ptr &= DISPATCH_QUEUE_ENQUEUED | DISPATCH_QUEUE_ENQUEUED_ON_MGR;
    *owned_ptr |= owned;
    if (unlikely(flags & DISPATCH_INVOKE_DISALLOW_SYNC_WAITERS)) {
        DISPATCH_INTERNAL_CRASH(dc,
                "Deferred continuation on source, mach channel or mgr");
    }
    _dispatch_thread_frame_pop(&dtf);
    return dq->do_targetq;
}

static inline void _dispatch_continuation_pop_inline(dispatch_object_t dou,
        dispatch_invoke_context_t dic, dispatch_invoke_flags_t flags,
```

```
        dispatch_queue_t dq)
{
    dispatch_pthread_root_queue_observer_hooks_t observer_hooks =
            _dispatch_get_pthread_root_queue_observer_hooks();
    if (observer_hooks) observer_hooks->queue_will_execute(dq);
    _dispatch_trace_continuation_pop(dq, dou);
    flags &= _DISPATCH_INVOKE_PROPAGATE_MASK;
    if (_dispatch_object_has_vtable(dou)) {
        dx_invoke(dou._do, dic, flags);
    } else {
        _dispatch_continuation_invoke_inline(dou, DISPATCH_NO_VOUCHER, flags)
;
    }
    if (observer_hooks) observer_hooks->queue_did_execute(dq);
}

static inline void _dispatch_continuation_invoke_inline(dispatch_object_t dou
, voucher_t ov,
        dispatch_invoke_flags_t flags)
{
    dispatch_continuation_t dc = dou._dc, dc1;
    dispatch_invoke_with_autoreleasepool(flags, {
        uintptr_t dc_flags = dc->dc_flags;
        _dispatch_continuation_voucher_adopt(dc, ov, dc_flags);
        if (dc_flags & DISPATCH_OBJ_CONSUME_BIT) {
            dc1 = _dispatch_continuation_free_cacheonly(dc);
        } else {
            dc1 = NULL;
        }
        if (unlikely(dc_flags & DISPATCH_OBJ_GROUP_BIT)) {
            _dispatch_continuation_with_group_invoke(dc);
        } else {
            // 调用_dispatch_client_callout执行block
            _dispatch_client_callout(dc->dc_ctxt, dc->dc_func);
            _dispatch_introspection_queue_item_complete(dou);
        }
        if (unlikely(dc1)) {
            _dispatch_continuation_free_to_cache_limit(dc1);
        }
    });
    _dispatch_perfmon_workitem_inc();
}
```

0x10 dispatch_get_global_queue

可以发现，dispatch_get_global_queue其实就是取对应优先级的root队列拿来用。所以上面也

提过，为啥在global_queue里面不能用barrier。

```
dispatch_get_global_queue(long priority, unsigned long flags)
{
    if (flags & ~(unsigned long)DISPATCH_QUEUE_OVERCOMMIT) {
        return DISPATCH_BAD_INPUT;
    }
    dispatch_qos_t qos = _dispatch_qos_from_queue_priority(priority);

    if (qos == DISPATCH_QOS_UNSPECIFIED) {
        return DISPATCH_BAD_INPUT;
    }
    return _dispatch_get_root_queue(qos, flags & DISPATCH_QUEUE_OVERCOMMIT);
}

static inline dispatch_queue_t _dispatch_get_root_queue(dispatch_qos_t qos, b
ool overcommit)
{
    if (unlikely(qos == DISPATCH_QOS_UNSPECIFIED || qos > DISPATCH_QOS_MAX))
{
        DISPATCH_CLIENT_CRASH(qos, "Corrupted priority");
    }
    return &_dispatch_root_queues[2 * (qos - 1) + overcommit];
}
```