

BeeHive

BeeHive核心思想涉及两个部分：

- 模块和服务以注册方式来达到对模块和服务的集中式管理思想
 - 统计管理初始化流程，注册的模块可以按照优先级排序，谁先谁后初始化，
 - 为所有注册的模块设置统一的处理方式，`setup`, `init`, 全部有30中event都可以统一处理
 - `Core+Plugin`的形式可以让一个应用主流程部分得到集中管理，不同模块以plugin形式存在，便于横向的扩展和移植
- 各个模块间解耦，从直接调用对应模块，变成以Service的形式，避免了直接依赖。
 - service注册方式，让模块解耦，让服务可视化，同时这样做也为组件化提供了非常有利的条件

注册流程

- 各个模块和service都可以使用三个方式注册到BeeHive模块中进行管理

模块注册

- 模块注册有BHManager进行管理，在注册过程中会创建模块的实例对象，在加入管理的容器中会对模块进行优先级的排序
- 其中+load方法注册早于dyld方法注册，两个方式都在main之前
- 本地加载plist注册在main之后注册
- 所有注册的方式都在第一屏显示前注册成功
- 所有模块的触发两个Event, `setup`和`init`都会在第一屏展示前准备好
- 设计三种不同的方式注册，我的理解主要目的还是为了加速启动流程，分批注册module和service

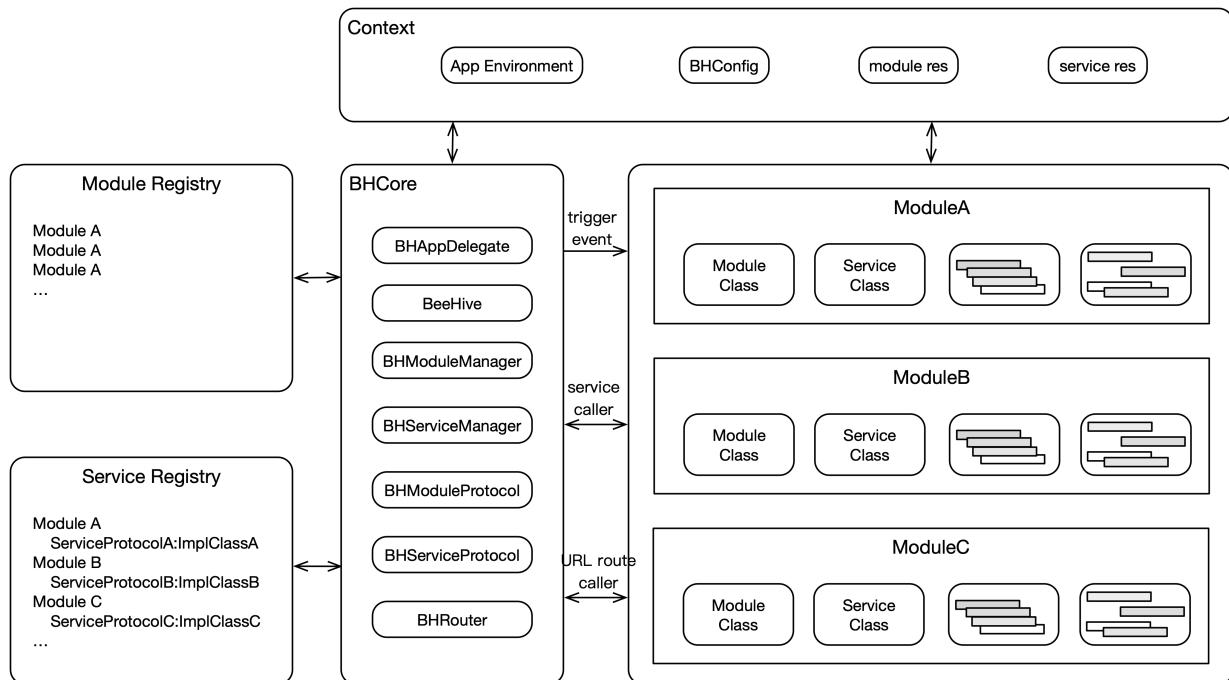
service注册

- service注册比较简单，只是管理seervice和serviceimpl的映射关以及如何通过service获取serviceimpl

- serviceimpl只有在获取时候才会被实例化，就算是单例也是在获取第一次的时候实例化，base service protocol只有两个方法声明，是否是单例，如果是单例需要提供shareinstance方法
- module

BeeHive概览

BeeHive的架构图如下所示：



图中的BHContext，包含BeeHive的配置文件，提供全局统一上下文信息。

图中的BHCore包含：

- BeeHive， 提供组件库对外接口
- BHModuleManager和BHModuleProtocol， 注册和创建Module逻辑
- BHServiceManager和BHServiceProtocol， 注册和创建Service逻辑
- BHRouter

Module、Service注册和调用逻辑只和核心模块相关，Module之间没有直接的关联关系。

BeeHive模块注册

使用注解的方式注册Module和Service时，Module和Service的注册发生在加载镜像文件时期。

plist方式注册Module和Service，是在AppDelegate中设置BeeHive的Context时加载注册。

Module动态注册是在+load方法中，也是在加载镜像时注册。Service的动态注册可以Module的modInit:或者modSetup:中，或者使用时注册。

模块注册有三种方式：Annotation方式注册、读取本地plist方式注册、Load方法注册。所谓注册，就是将Module类告知BHModuleManager来管理。由此可见，在BeeHive中是通过BHModuleManager来管理各个模块的，BHModuleManager中只会管理已经被注册过的模块。

Module、Service之间没有关联，每个业务模块可以单独实现Module或者Service的功能。

Annotation方式注册

通过BeeHiveMod宏进行Annotation标记.注解是在编辑阶段完成的。

```
BeeHiveMod(ShopModule)

//BeeHiveMod宏定义如下:
#define BeeHiveMod(name) \
class BeeHive; char * k##name##_mod BeeHiveDATA(Beehi
veMods) = ""#name"";

//BeeHiveDATA也是一个宏:
#define BeeHiveDATA(sectname) __attribute__((used, sect
ion("__DATA,#sectname"")))

//在预编译结束后, BeeHiveMode宏最终会完全展开成下面的样子:
class BeeHive; char * kShopModule_mod __attribute__((us
ed, section("__DATA,""BeehiveMods""))) = """ShopModu
le""";
```

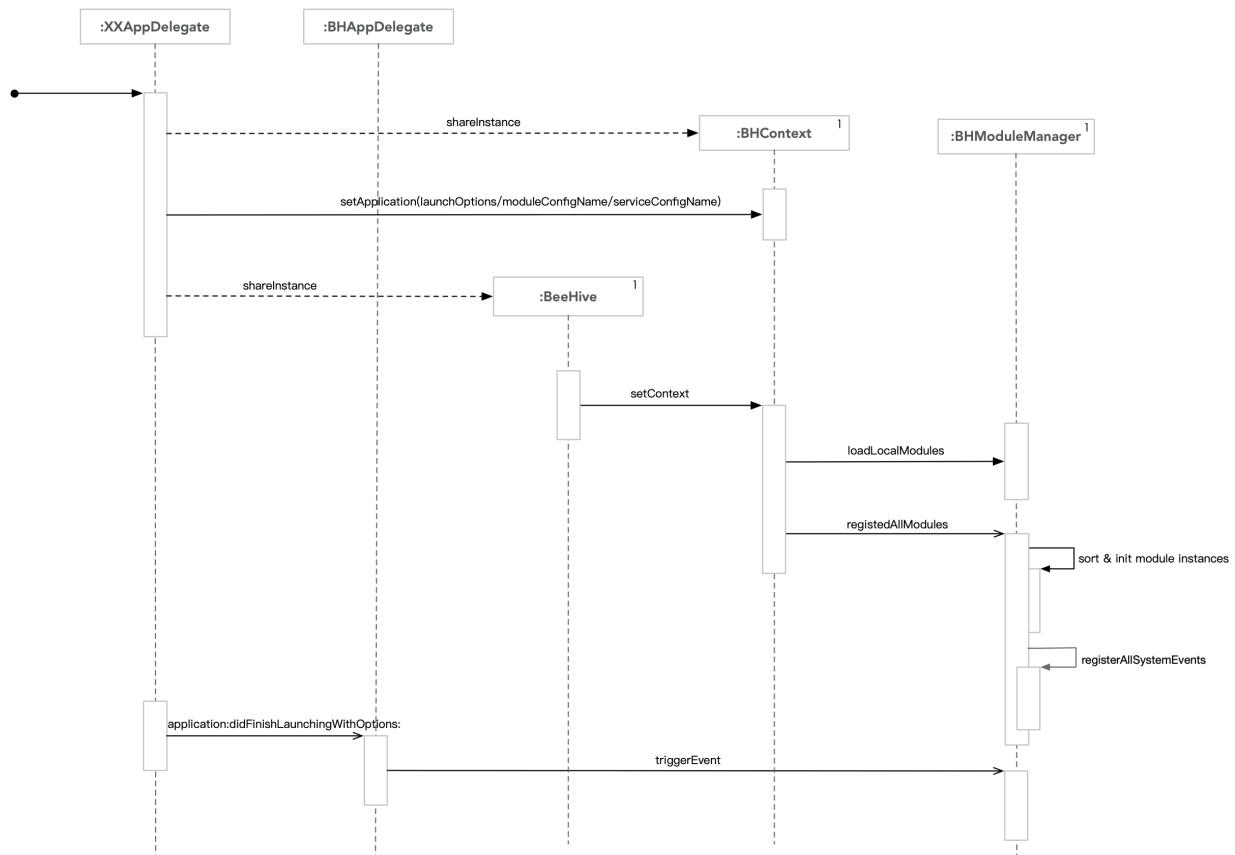
关于__attribute的用法，可参考我的另一篇attribute

__attribute第一个参数used,它的作用是告诉编译器，我声明的这个符号是需要保留的。被used修饰以后，意味着即使函数没有被引用，在Release下也不会被优化。如果不加这个修饰，那么Release环境链接器会去掉没有被引用的段。

有时候我们需要指定一个特殊的段，来存放我们想要的数据。这里我们就把数据存在__DATA数据段里面的“BeehiveMods”section中。Attributes的修饰关键字

section (“section-name”)可以达到此要求。

上述代码中：

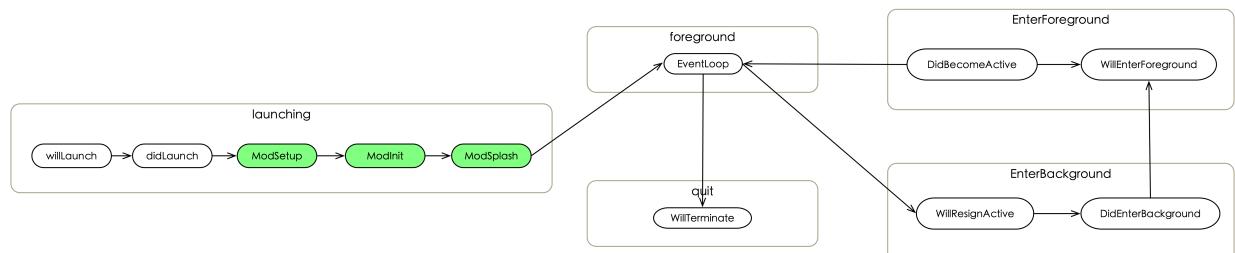


图中包含了主要的BeeHive启动过程以及Module注册的时序逻辑。Module的事件分发源于BHAppDelegate中的triggerEvent。

加载Module：

- BeeHive.plist中配置的module和service是在 AppDelegate中调用 `[[BeeHive shareInstance] setContext:[BHContext shareInstance]]`; 时加载。
- Module的实现中 `+load`内部调用 `[BeeHive registerDynamicModule:[self class]]`; 动态加载。
- Module的实现中使用注解： `@BeeHiveMod(XXModule)`

BHAppDelegate中除了回调系统的事件，还将App生命周期进行扩展，增加 `ModuleSetup`, `ModuleInit`, `ModuleSplash`, 此外开发人员还可以自行扩展。



Module遵循BHModuleProtocol后，能够捕获App状态的回调，并拥有App生命周

期内的全局上下文，通过context可获取配置参数，模块资源以及服务资源。

以BeeHive作为底层框架的App，除了解耦带来的便利，开发人员在开发新App过程中涉及相同功能的Module，无需重复造轮子，直接移植Module，开发一个App如同拼装积木，能组合需要的功能业务。