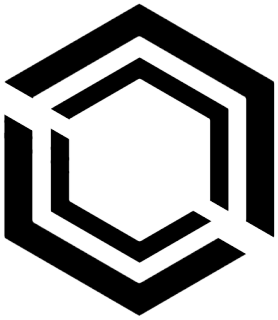


BEE Framework

App 3.0 技术架构的先行者

老郭和他的小伙伴们
gavinkwoe@gmail.com

谨以此献给关注和使用 {Bee} 的朋友们。

- 老郭和他的小伙伴们

目录

写在前面

随着 Three20 的倒下及 Nimbus 的崛起，两年之间，巨头 Facebook 与 Google 完成了“顶级框架”的交接棒，由一群来自于世界各地各公司的一线开发者组成的开源团队推动着整个行业的技术革新，而其他百万开源使用者直接或间接收益其中。Github 社区的出现让我们看到，开发者并非商品，价值并非等价于金钱，在这样平坦的世界里合作的结果，往往是催生更多使用开源而具有商业价值的产品出现。{Bee} 作者有幸参与其中，不仅使用开源，而且创造开源，乐此不疲，受益匪浅。

回到移动开发这些事，不管你属于哪一阵营，是做 Native 开发，Web 开发，或是 Hybrid 开发，任何一种技术对开发者来说本质是一样的，适者生存是自古以来的道理。争议较多的无非是 Web 在模仿 Native 效果将用到大量代码，效果不一定与 Native 一样好，而 Native app 本身开发及调试的效率又不尽人意。这时，必然会有人挺身而出，拨剑起而，喊着“融合起来”的口号，带来了新的技术方案。

{Bee} 介于 Native 与 Web 两者之间，致力于解决 App 开发的效率问题，希望能够从技术上真正打破现状。区别于 Hybrid 开发思想，我们为开发者提供了更好的与 Native 开发结合的方式，是一种保留 Web 前端开发优势的混合型开发技术，一边采用高集成度的 Native 框架完成逻辑及存储部分，一边用优化后的 Web 前端框架来完成界面部分。

{Bee} 框架采用最为开放的 MIT 协议，承诺长期保持免费，如果你认可我们技术，您可以使用 {Bee} 在任何一款商业化软件中，并且可以免费得到我们的技术支援。

相信在将来，随处可见带有 {Bee} 标识的产品出现，它代表着至今我们所想的一切：

打破层层壁垒的开放与颠覆！

什么是 App 3.0?

iOS 发布的 5 年里，作为开发者，我们见证了太多顶级公司对于尝试改变现有开发模式的技术出现。它们就像长相精致的“女神”，第一眼远观养眼的美丽，不可接近，第二眼天生丽质，近看不出化妆的痕迹。而生活中，我们更需要“心更近”的第三眼美女的出现。它们像杯清茶，入口清谈，回味醇香，没有太多花哨的包装，实在，体贴，包容，内秀，不妄想，不脱离现实，随着日子的流逝，扎根到了你的生活中。

也许你听过或用过 PhoneGap，也许听过或用过 AdobeAir，也许听过或用过 RubyMotion，不管怎样，你正被夹在 Native or Web 的交叉路口，Web 化是的一条探索中的道路，但不是终点。

如果将前一代 Hybrid 开发技术归为 App 2.0 技术架构（即 Native + HTML + CSS），那么介于前一代技术中的痛点，App 3.0 新的架构为我们带来更高的开发效率，更适合于移动化，总结它的特点：

StrongNative + MobileHTML + MobileCSS

1. View 模版化
    a. 具有语意的简化的 HTML 标签
    b. 针对移动端优化的 CSS 样式
    c. 将出现开源的模版样式标准库
2. Model 对象化
    a. 本地存储对象化（ActiveRecord 的特性引入）
    b. 存储关系对象化（通过类继承或嵌套直接表示）
3. Controller 接口化
    a. 协议解析对象化（JSON 与 NSObject 相互转换）
    b. 提供自动生成工具（基于 JSON Schema 的代码生成）

这意味着 Native 与 Web 技术之间真正的融合，将成为新一代顶级移动端技术框架的衡量标准。

{Bee} 的起源

Bee 原始版本产生于手机 QQ 空间 iPhone 版本，最初解决层与层之间的简单解耦问题，随作者对其不断的改进，至今已发展到 0.4.0 版本。

Bee 是 App 3.0 技术架构的先行者，基于 MVC 架构，适用于大规模 App（信息展示、 社交、工具等类型）的快速开发框架，已被诸多企业的产品采用，如：腾讯，新浪等公司。

{Bee} 的特点

1. 使用简单
2. 高度解耦　(Pure MVC)
3. 模版化　(XML template)
4. 自动生成　(Scaffolding)
5. 内建工具　(In-App Debugger)
6. 社区庞大　(INSTHUB community)

谁应该使用 {Bee}？

1. 我想快速完成开发
2. 我讨厌编写冗长的代码
3. 我总是喜欢找点乐子
4. 我的项目需求大的吓人，不知从何入手

基本要求

了解 Objective-C & Cocoa-Touch
了解 MVC
了解 XML，CSS

{Bee} 对比 Nimbus

	Nimbus	{Bee}
View - Animation	✖	✔
View - Container	✖	✔
View - CSS	✔	✔
View - Data binding	✖	✔
View - Signal	✖	✔
View - Layouter	✖	✔
View - Query	✖	✔
View - Template	✖	✔
View - Cell	✖	✔
Controller - Controller	✖	✔
Controller - Message	✖	✔
Controller - Routine	✖	✔
Model - File cache	✖	✔
Model - Memory cache	✔	✔
Model - Keychain	✖	✔
Model - User defauls	✖	✔
Model - Networking	✖	✔
Model - Database	✖	✔
Model - ActiveObject	✖	✔
Model - ActiveRecord	✖	✔
Model - Table model	✔	✖
Foundation - Runtime	✖	✔
Foundation - Performance	✖	✔
Foundation - Log	✔	✔
Foundation - Assert	✖	✔
Service - Debugger	✖	✔
Service - Inspector	✖	✔
Service - LBS	✖	✔
Service - APNS	✖	✔
Other - Camera	✖	✔
Other - MatrixView	✖	✔
Other - MultiLineScroll	✖	✔
Other - TipsView	✖	✔

Other - ZoomView	✖	✔
Other - Attributed label	✔	✖
Other - Launcher	✔	✖
Other - Web view	✔	✔
Other - Badge	✔	✖
Other - Inter-app	✔	✖
Tools - Scaffold	✖	✔

不用多说，对于一个框架级产品来讲，{Bee}的完整度要高出很多。

新手入门 - 下载安装

环境要求

Mac OS 10.7 及以上版本
XCode 4.6 及以上版本

如何安装

1. 打开 http://www.github.com/gavinkwoe/beeframework
2. 下载 Download zip 后解压
3. 再将/framework 目录拖拽进工程
4. 在您的工程中记得#import “bee.h”即可开始使用

运行 Demo

1. 双击/projects/BeeFramework.xcworkspace
2. 选择 example > iPhone simulator
3. 编译并运行

新手入门 - 目录结构

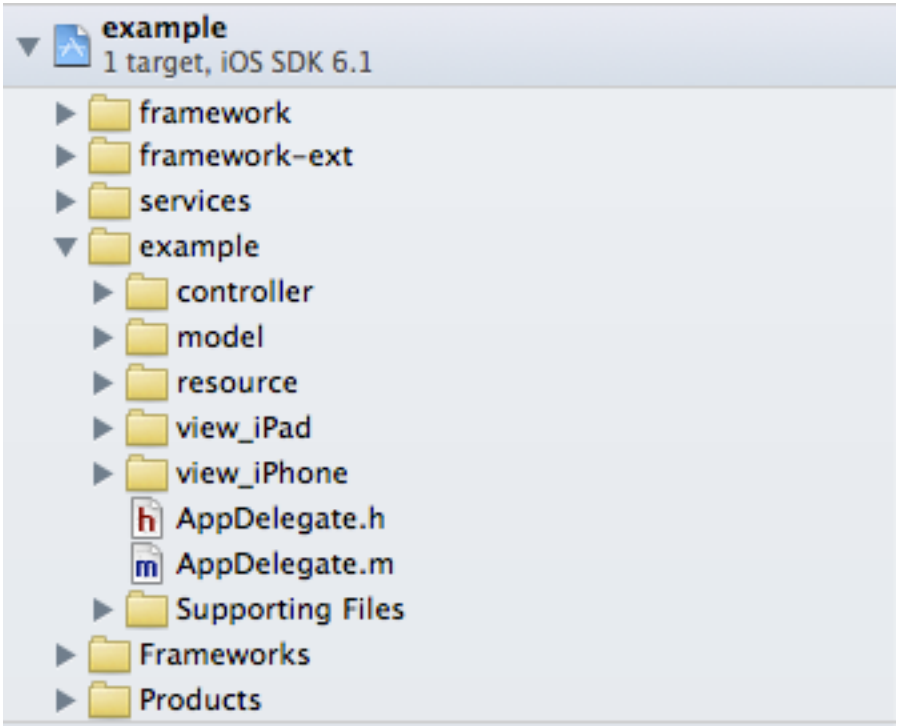
\coding-template	编码规范
\deprecated	老代码（不建议使用）
\design	相关设计
\documents	相关文档
\framework	框架主目录
\application	应用程序
\service	基础服务
\system	系统组件
\vendor	第三方库
\framework-ext	框架扩展
\projects	示例工程
\lib	编译.lib
\example	例子 APP
\services	扩展服务
\ServiceDebugger	应用内调试器
\ServiceInspector	应用内观察器
\ServiceLocation	位置服务
\ServicePush	推送服务
\tools	工具
\scaffold	代码生成器

新手入门 - 工程结构

以 DEMO 工程为例，一起来了解一下典型的 {Bee} App 工程怎样组织起来的。

1. 打开 projects/ BeeFramework.xcworkspace
2. 选择 Example/iPhone simulator 6.0
3. 编译并运行

可以看到，随着炫酷的动画，demo 程序运行起来了。
好，回到工程，我们可以看到，基本结构如下图：



其中 framework, framework-ext, services 属于 {Bee} 核心，先不必理会使具体实现。

用户代码存放于 example 目录中，包括下面几个部分：

controller - 业务逻辑的代码存放于此

model	-	数据模型的代码存放于此
view_iPad	-	为 iPad 设备适配的视图模版及资源
view_iPhone	-	为 iPhone 设备适配的视图模版及资源
Supporting Files	-	工程设置相关
AppDelegate.h/.m	-	程序入口

建议开发者在使用 {Bee} 框架的同时，遵循相同的工程结构。

新手入门 - 编译选项

参见 bee_precompile.h，修改该文件来全局配置 {Bee} 功能特性。

__BEE_DEVELOPMENT__	是否开启开发模式（上线时请关闭）
__BEE_LOG__	是否开启日志（上线时请关闭）
__BEE_ASSERT__	是否开启断言（上线时请关闭）
__BEE_PERFORMANCE__	是否开启性能测试（上线时请关闭）
__BEE_UNITTEST__	是否开启单元测试（上线时请关闭）
__BEE_MOCKSERVER__	是否开启本地服务器（上线时请关闭）
__BEE_WIREFRAME__	是否开启线框图（上线时请关闭）

修改该文件将导致整个工程重新编译，除上面提到的编译选择外，其他请不要擅自改动，以免造成功能失效。

快速预览 - 编写程序入口

参考源码:

example/AppDelegate.h
example/AppDelegate.m
example/view_iPhone/AppBoard_iPhone.h
example/view_iPhone/AppBoard_iPhone.m
example/view_iPad/AppBoard_iPad.h
example/view_iPad/AppBoard_iPad.m

做为典型的程序入口，AppDelete 继承于 BeeSkeleton，根据 iPhone/iPad 两种分辨率，可以分别加载不同的视图，代码如下:

@implementation AppDelegate

```
- (void)load
{
	if ( [BeeSystemInfo isDevicePad] )
	{
	    self.window.rootViewController = [AppBoard_iPad sharedInstance];
	}
	else
	{
	    self.window.rootViewController = [AppBoard_iPhone sharedInstance];
	}
}
```

@end

除以上代码，还需要修改 C main:

```
	int main( int argc, char * argv[] )
	{
	     UIApplicationMain( argc, argv, nil @"AppDelegate" );
	}
```

好了，程序入口准备好了，接下来看看如何编写视图代码。

快速预览 - 编写视图结构

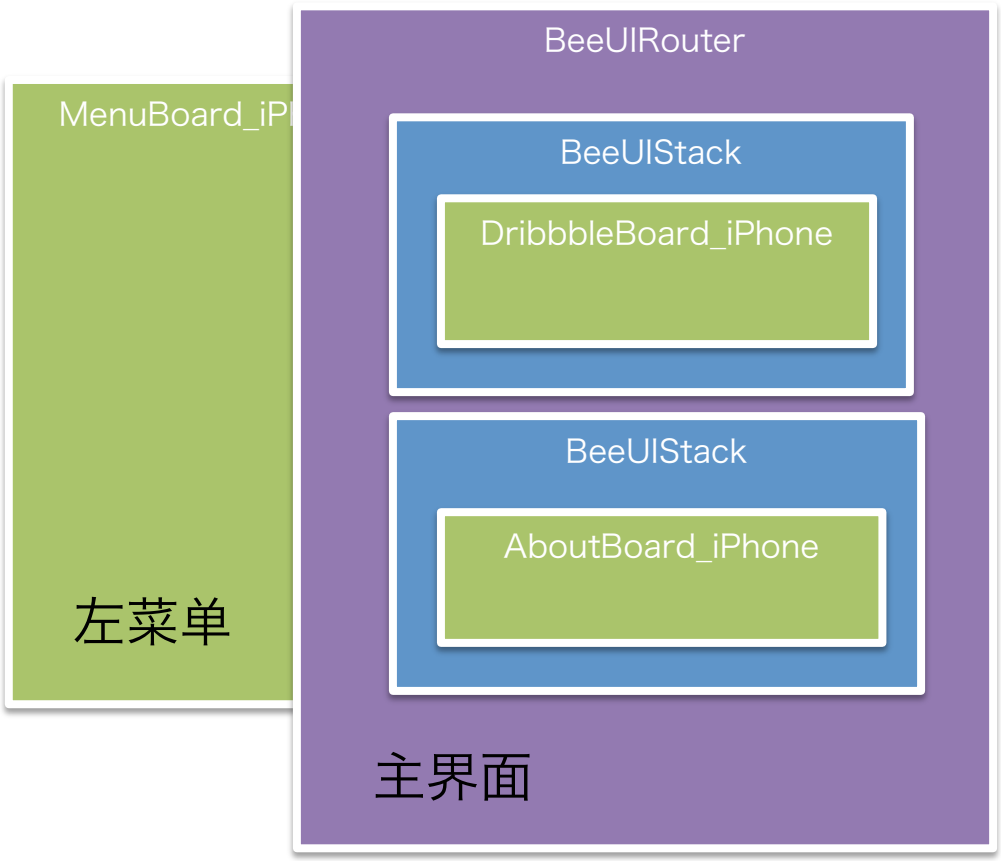
参考源码：

example/view_iPhone/AppBoard_iPhone.h
example/view_iPhone/AppBoard_iPhone.m

名词解释：

1. BeeUIBoard，　视图白板，控件的容器。
2. BeeUIStack，　视图堆栈，Board 的容器。
3. BeeUIRouter，　视图路由，Stack 的容器。

Demo 工程中，我们实现了类 Path 抽屉式的界面结构，如下图所示：



1）根据上图，创建视图结构的代码

```
_menu = [MenuBoard_iPhone sharedInstance];
...
[self.view addSubview:_menu.view];

_router = [BeeUIRouter sharedInstance];
...
[_router map:@"demo" toClass:[DribbbleBoard_iPhone class]];
[_router map:@"tutor" toClass:[TutorialBoard_iPhone class]];
[_router map:@"team" toClass:[TeamBoard_iPhone class]];
[_router map:@"about" toClass:[AboutBoard_iPhone class]];
[self.view addSubview:_router.view];
```

2）处理来自菜单的事件与主视图切换

```
ON_SIGNAL3( MenuBoard_iPhone, tutor, signal )
{
    [_router open:@"tutor" animated:YES];

    ...
}

ON_SIGNAL3( MenuBoard_iPhone, demo, signal )
{
    [_router open:@"demo" animated:YES];

    ...
}

ON_SIGNAL3( MenuBoard_iPhone, about, signal )
{
    [_router open:@"about" animated:YES];

    ...
}

ON_SIGNAL3( MenuBoard_iPhone, team, signal )
{
    [_router open:@"team" animated:YES];

    ...
}
```

接下来实现菜单及主界面视图。

快速预览 - 编写一个 View

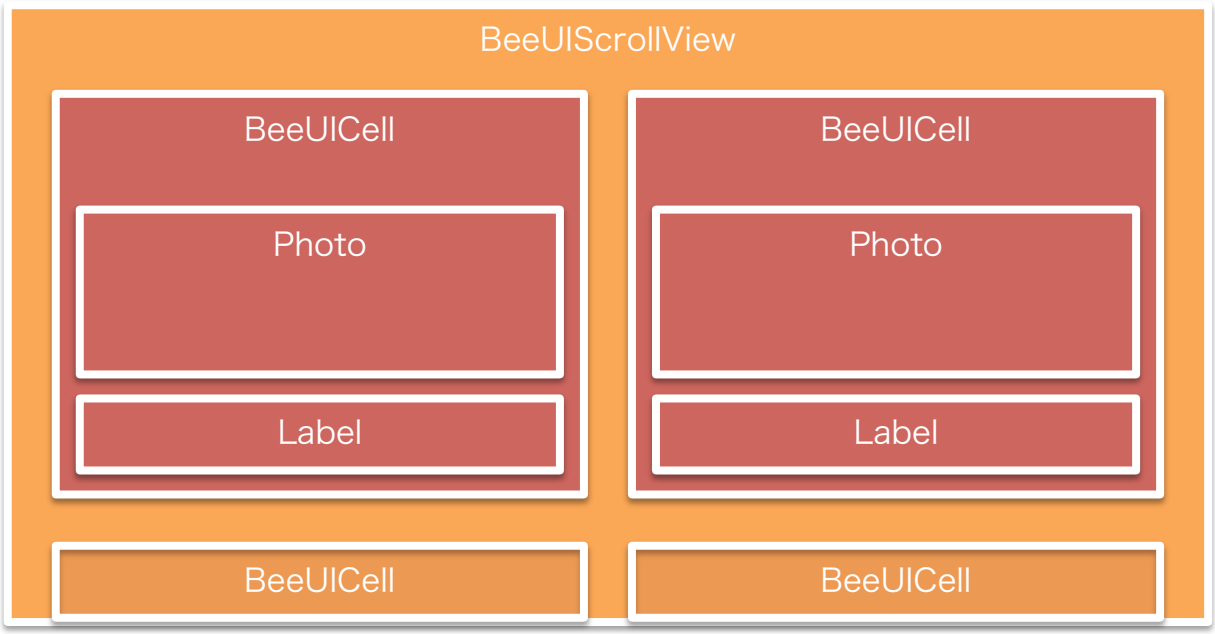
参考源码：

example/view_iPhone/templates/DribbbleBoard_iPhone.h
example/view_iPhone/templates/DribbbleBoard_iPhone.m

名词解释：

1. BeeUICell, 支持自动布局的格子控件
2. BeeUIScrollView, 支持多行多列的列表
3. BeeUISignal, 事件传递方式，见后面章节

接下来我们要实现一个简单的列表界面，样式如下：



1）定义 Cell 类（.h/.m）

```
@interface MyCell : BeeUICell
AS_SIGNAL( TOUCHED )
@end

@implementation MyCell

SUPPORT_AUTO_LAYOUT( YES )
```

```
SUPPORT_RESOURCE_LOADING( YES )

@end
```

2）定义 Cell 布局（.xml）

```
<?xml version="1.0" encoding="UTF-8"?>
<ui namespace="MyCell">
    <linear orientation="v" class="wrapper">
     <image id="photo" class="photo"/>
     <label id="label" class="label"/>
     …
    </linear>
    <style type="text/css">
     ...
    </style>
</ui>
```

3）定义 Board

```
@interface MyBoard : BeeUIBoard
@end

@implementation MyBoard
{
    BeeUIScrollView * _scroll;
}

- (void)handleUISignal_BeeUIBoard:(BeeUISignal *)signal
{
    [super handleUISignal:signal];

    if ( [signal is:BeeUIBoard.CREATE_VIEWS] )
    {
     _scroll = [[BeeUIScrollView alloc] init];
     _scroll.dataSource = self;
     _scroll.vertical = YES;
     [self.view addSubview:_scroll];
    }
    else if ( [signal is:BeeUIBoard.DELETE_VIEWS] )
    {
     SAFE_RELEASE_SUBVIEW( _scroll );
    }
    else if ( [signal is:BeeUIBoard.LAYOUT_VIEWS] )
    {
     _scroll.frame = self.viewBound;
    }
    else if ( [signal is:BeeUIBoard.WILL_APPEAR] )
```

```
        {
         [_scroll reloadData];
        }
    }
```

4）实现 DataSource

```
    // 一共多少列
    - (NSInteger)numberOfLinesInScrollView:(BeeUIScrollView *)scrollView
    {
        return 2;
    }

    // 一共多少行
    - (NSInteger)numberOfViewsInScrollView:(BeeUIScrollView *)scrollView
    {
        return 30;
    }

    // 为每个单元格创建Cell
    - (UIView *)scrollView:(BeeUIScrollView *)scrollView
              viewForIndex:(NSInteger)index
    {
        MyCell * cell = [scrollView dequeueWithContentClass:[MyCell
    class]];
        cell.data = …;
        return cell;
    }

    // 计算每个单元格宽高
    - (CGSize)scrollView:(BeeUIScrollView *)scrollView
    sizeForIndex:(NSInteger)index
    {
        return [MyCell estimateUISizeByWidth:scrollView.width
                                     forData:...];
    }
```

6）响应 Signal 事件

```
    ON_SIGNAL3( MyCell, TOUCHED, signal )
    {
        [super handleUISignal:signal];

        if ( [signal is:MyCell.TOUCHED] )
        {
               MyCell * cell = signal.source;
```

```
                … …
        }
    }

    ON_SIGNAL3( MyCell, ..., signal )
    {
        [super handleUISignal:signal];
    }
```

到这里，一个 View 的雏形展示出来了，更多代码详见 Demo 工程。

快速预览 - 编写一个 Controller

在{Bee}的世界里，所有的业务逻辑都封装在 Controller 里，每条逻辑可用 Named Message 来命名和进行异步调用，引入几个名词：

BeeMessage: 消息，即“逻辑”调用时的事件载体
BeeController: 控制器，即“逻辑”的容器

还是以 Ruby-China 为例，nodes.json 的 API 接口实现如下：
http://ruby-china.org/api/nodes.json

1）声明 Controllert 和 Message

```
@interface RubyChinaAPI : BeeController
AS_MESSAGE( GET_NODES );
@end
```

2）实现 Controller 和 Message

```
@implementation RubyChinaAPI

DEF_MESSAGE( GET_NODES );

- (void)GET_NODES:(BeeMessage *)msg
{
    if ( msg.sending )
    {
        msg.HTTP_GET(@"http://ruby-china.org/api/nodes.json" );
    }
    else if ( msg.succeed )
    {
        NSArray * result = msg.responseJSONArray;
        if ( nil == result )
        {
            msg.failed = YES;
            return;
        }

        msg.OUTPUT( @”result”, result );
    }
    else if ( msg.failed )
    {
```

```
    }
}

@end
```

3）通过 NamedMessage 调用 Controller

```
@implementation MyBoard

- (void)handleUISignal_BeeUIBoard:(BeeUISignal *)signal
{
    ...
    else if ( [signal is:BeeUIBoard.WILL_APPEAR] )
    {
     self.MSG( RubyChinaAPI.GET_NODES );
    }
}

@end
```

4）处理 NamedMessage 调用结果

```
@implementation MyBoard

- (void)handleMessage:(BeeMessage *)msg
{
    [super handleMessage:msg];

    if ( [msg is:RubyChinaAPI.GET_NODES] )
    {
     if ( msg.succeed )
     {
          NSArray * result = msg.GET_OUTPUT( @”result” );
          ...

          [_scroll reloadData];
     }
    }
}

@end
```

注：{Bee} 0.4.0 及以后版本可以通过 scaffold 工具自动生成，使用方法请查看/tools/scaffold.md。

快速预览 - 编写一个 Model

在{Bee}的世界里，几乎所有的数据结构都能用 ActiveRecord 表示，并将复杂重复的 CRUD 操作封装在 Model 中，引入几个名词：

BeeDataBase:	数据库，对 SQLITE 的封装
BeeActiveObject:	活动对象，支持基本的序列化反序列化
BeeActiveRecord:	活动记录，对象化的 SQL 数据库记录
BeeModel:	数据模型，用来封装 AR 的增删改查

以 Ruby-China 为例，nodes.json 接口返回的数据如下：
http://ruby-china.org/api/nodes.json

```
[
    {
        "id":1,
        "name":"Ruby",
        "topics_count":841,
        "summary":"Ruby",
        "section_id":1,
        "sort":0,
        "section_name":"Ruby"
    },
    ...
]
```

1）定义 ActiveRecord

```
@interface RubyChinaNode : BeeActiveRecord
@property (nonatomic, retain) NSNumber *    id;
@property (nonatomic, retain) NSString *    name;
@property (nonatomic, retain) NSNumber *    topics_count;
@property (nonatomic, retain) NSString *    summary;
@property (nonatomic, retain) NSNumber *    section_id;
@property (nonatomic, retain) NSString *    section_name;
@property (nonatomic, retain) NSNumber *    sort;
@end

@implementation RubyChinaNode
@end
```

2）常用 ActiveRecord 增删改查

```
RubyChinaNode * n = [RubyChinaNode record:…];
RubyChinaNode * n = [RubyChinaNode recordWithKey:…];
RubyChinaNode * n = [RubyChinaNode recordWithObject:…];
RubyChinaNode * n = [RubyChinaNode recordWithDictionary:…];
RubyChinaNode * n = [RubyChinaNode recordWithJSONData:…];
RubyChinaNode * n = [RubyChinaNode recordWithJSONString:…];

NSArray * array = …;
RubyChinaNode.DB.SAVE( array );

RubyChinaNode
.DB
.ORDER_ASC_BY( @"section_id" )
.ORDER_ASC_BY( @"sort" )
.GET_RECORDS();

RubyChinaNode
.DB
.WHERE( @"section_id", @”0” )
.GET_RECORDS();

RubyChinaNode
.DB
.WHERE( @"section_id", @”0” )
.COUNT();
```

3）获取 ActiveRecord 查询结果

```
if ( RubyChinaNode.DB.succeed )
{
      RubyChinaNode.DB.resultArray;    // 数据
      RubyChinaNode.DB.resultCount;    // 数量
}
```

4）根据需求编写 Model

```
@interface RubyChinaNodeModel : BeeModel

@property (nonatomic, retain) NSMutableArray * nodes;

- (void)getNodes;

@end

@implementation RubyChinaNodeModel

@synthesize nodes = _nodes;
```

```
- (void)load
{
     _nodes = [[NSMutableArray alloc] init];
}

- (void)unload
{
     [_nodes removeAllObjects];
     [_nodes release];
}

- (void)getNodes
{
     self.MSG( RubyChinaAPI.GET_NODES );
}

- (void)handleMessage:(BeeMessage *)msg
{
    [super handleMessage:msg];

    if ( [msg is:RubyChinaAPI.GET_NODES] )
    {
        if ( msg.succeed )
        {
           NSArray * result = msg.GET_OUTPUT( @"result" );

           [_nodes removeAllObjects];
           [_nodes addObjects:result];
        }
    }
}
```

注：{Bee} 0.4.0 及以后版本可以通过 scaffold 工具自动生成，使用方法请查看/tools/scaffold.md。

快速预览 - 整合起来

这里我们尝试跑通以上所有流程:
1. 从 ruby-china.com 拉取 nodes
2. 将 nodes 持久化到 SQL DB
3. 再将 nodes 显示出来

步骤 1 和 2 已经在前面章节中实现了，接下来我们只需要做两件事:

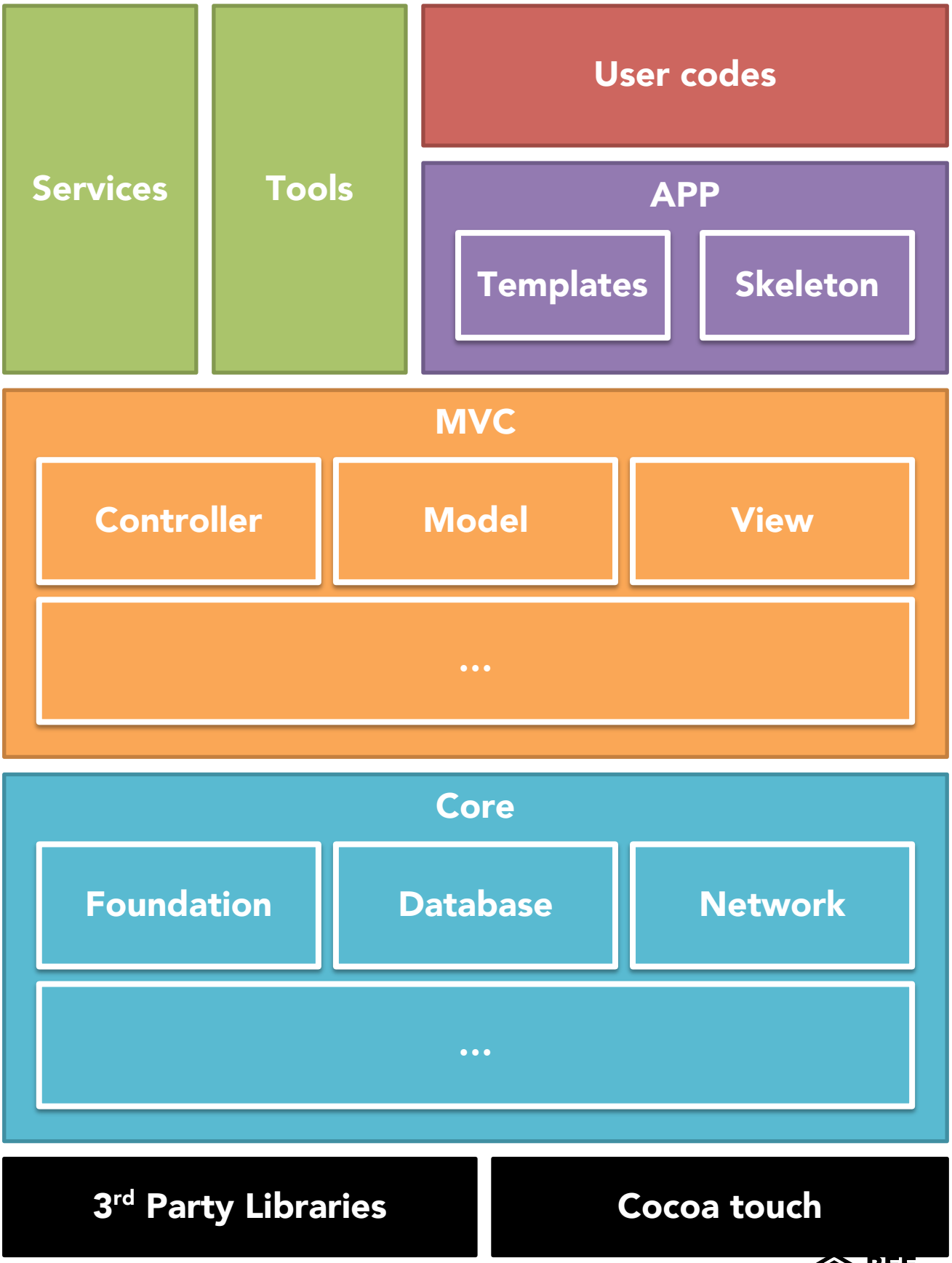
1. 调用 Model 拉取数据 nodes
2. 将 nodes 数据显示到 MyCell 上

```
@implementation MyBoard
{
    RubyChinaNodeModel * _model;
}

- (void)load
{
    [super load];

    _model = [[RubyChinaNodeModel alloc] init];
    [_model addObserver:self];
}

- (void)unload
{
    [_model removeObserver:self];
    [_model release];

    [super unload];
}

...

    else if ( [signal is:BeeUIBoard.WILL_APPEAR] )
    {
     [_model getNodes];
    }

...

- (void)handleMessage:(BeeMessage *)msg
```

```
{
    [super handleMessage:msg];

    if ( [msg is:RubyChinaAPI.GET_NODES] )
    {
        if ( msg.succeed )
        {
           [_scroll reloadData];
        }
    }
}

- (UIView *)scrollView:(BeeUIScrollView *)scrollView
          viewForIndex:(NSInteger)index
{
    MyCell * cell = …;
    cell.data = [_model.nodes objectAtIndex:index];
    return cell;
}

@end


@implementation MyCell

- (void)dataDidChanged
{
    RubyChinaNode * n = self.data;

    $(@"#label").DATA( n.section_name );
}

@end
```

到此为止，相信您已经大致了解 {Bee} 开发流程了，在后面的章节可以进一步学习 {Bee} 模块的原理及设计理念，建议在学习的同时，可以参考 Demo 工程编写一个自己的 App。

深入学习 - 理解整体架构


Services
Tools
User codes
APP
Templates
Skeleton
MVC
Controller
Model
View
...
Core
Foundation
Database
Network
...
3rd Party Libraries
Cocoa touch

基本架构分为五层：

1. OS 层：　　即 Cocoa touch，以及一些基础第三方库。
2. Core 层：　其中包括 基础 API、存储、网络、LOG、测试等组件。
3. MVC 层：　其中包括 控制器，消息通讯，样式/模版布局等组件。
4. App 层：　将 Core、MVC 封装进 App，并提供服务及工具。
5. User 层：　用户基于以上 4 层快速构建出自己的 App。

对比其他框架：

1. Three20：组件丰富，没有系统的架构，过于零散。
2. Nimbus：　组件丰富，没有系统的架构，过于零散。

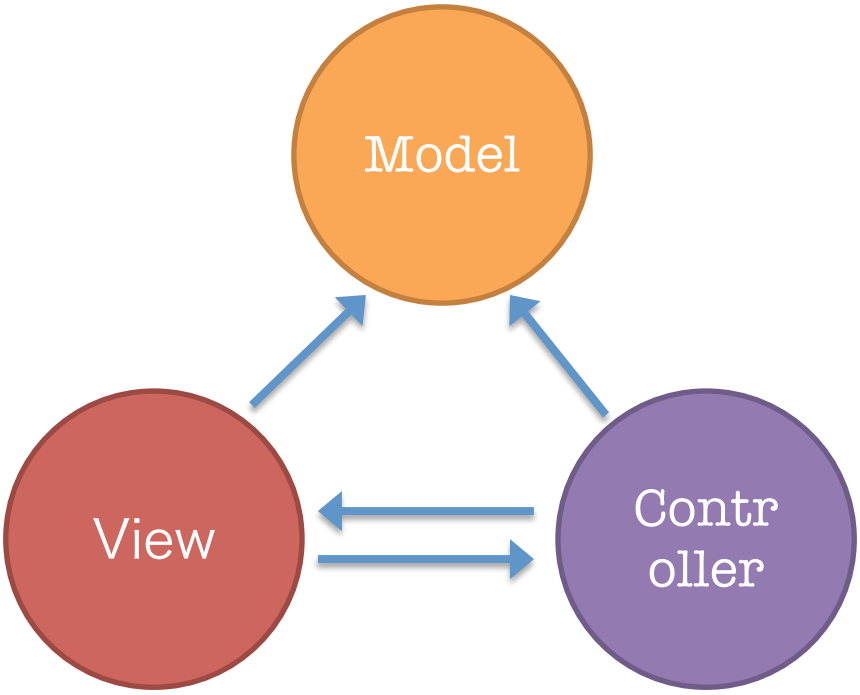
深入学习 {Bee} 的架构，会帮助您进一步提高自己对软件架构的认识。

深入学习 - 理解 MVC

参考源码：

Framework/application/mvc

从工程结构中可以清晰的看到，在 {Bee} 的世界里，我们把 App 按功能拆解为 Model-View-Controller 三个部分，他们的关系如下：



按照 MVC 模型，归纳存放你的代码。

1. View 视图

视图是用户看到并与之交互的界面。对老式（0.4 以前） {Bee} 的应用程序来说，视图就是 UIView 元素组成的界面，在新式的（0.4 以后） {Bee} 的应用程序来说，视图是 XML+CSS 元素组成的界面。

在 {Bee} 中 MVC-View 带来的好处是，不管给出的数据是表示一条新闻或是一张图片，均可以使用 View 模版来显示并与用户互动。

2. Model 模型

模型表示数据和规则。在 {Bee} 中，我们将其视为视图数据提供者。这里 Model 必须继承于 BeeModel，它提供了更好的通知机制，使其能够与 View 结合，做到一次编写被多个视图所用，减少重复性工作。

3. Controller 控制器

控制器表示着业务逻辑，通常接受用户的输入，调用网络接口获取数据，并调用 Model 存储，最后通知 View 显示数据。这里 Controller 必须继承于 BeeController，更好的一次编写逻辑被多 View 复用。

牢记 {Bee} 几个常用类，分别代表着 MVC 的关键角色。

1. BeeUIBoard - 视图控制器
2. BeeUIStack - 视图堆栈控制器
3. BeeUIRouter - 视图路由控制器
4. BeeController - 逻辑控制器
5. BeeMessage - 逻辑消息（View 与 Controller 之间）
6. BeeModel - 数据模型

后面的的章节，我们继续学习。

深入学习 - 理解 Skeleton

参考源码:

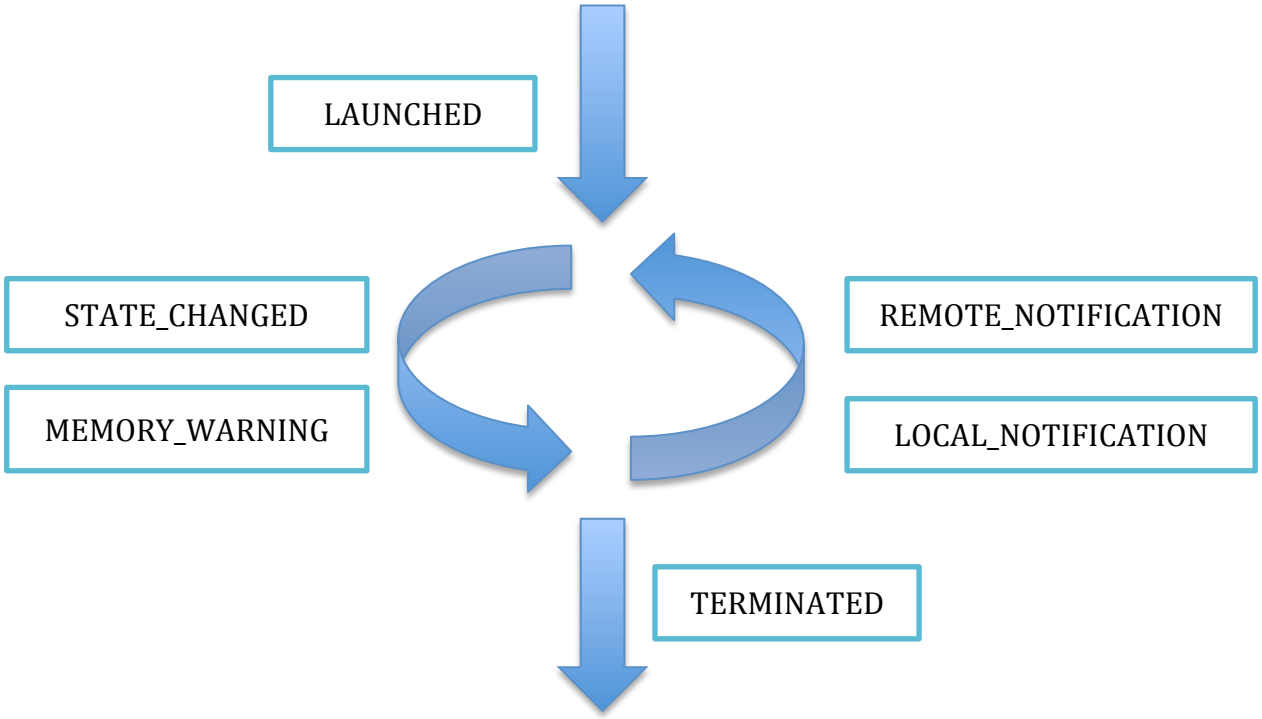
Framework/application/skeleton

顾名思义，BeeSkeleton 做为程序的骨架，负责整个 App 的入口以及生命期管理。从 0.4.0 以后版本，建议开发者使用它做为应用程序的默认入口。

BeeSkeleton 提供了哪些功能?

1. 管理 App 生命期
2. 管理状态变化通知

BeeSkeleton 封装了 App 流程，在状态转化时会发出通知，如下图:



我们来看一下具体用法,

```
@interface AppDelegate : BeeSkeleton
@end

@implementation AppDelegate

- (void)load
{
      self.window.rootViewController = <第一个用户界面入口>;
}

@end
```

另外还需要修改一下 main.m:

```
{
      UIApplicationMain( argc, argv, nil @”AppDelegate” );
}
```

深入学习 - 理解 View

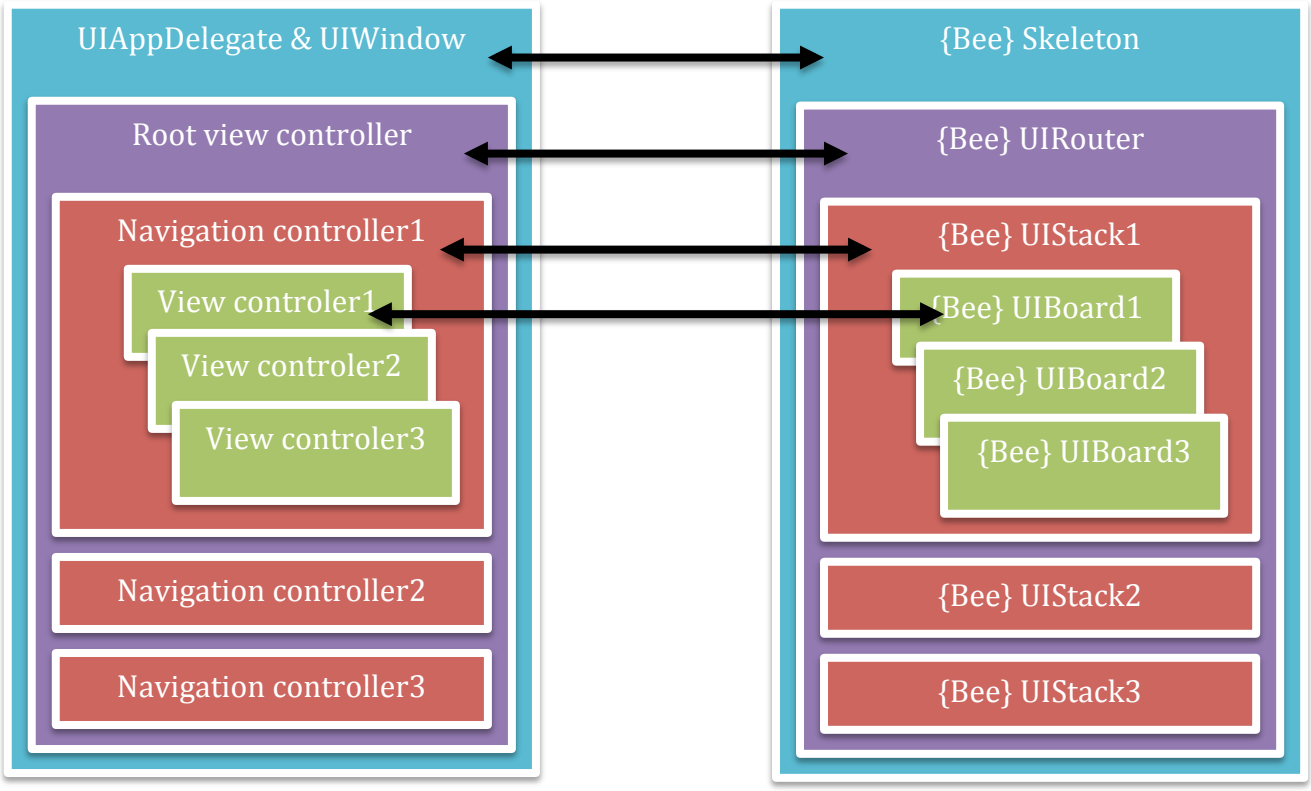
参考源码:

Framework/application/mvc/view

这一章节的内容比较多，按以下顺序依次讲解:

1. {Bee} 提供了哪些控件?
2. {Bee} 提供了哪些容器?
3. {Bee} 如何编写界面布局?
4. {Bee} 如何改变界面样式?
5. {Bee} 如何填写界面数据?
6. {Bee} 如何响应事件?

对比 View 的 4 层结构，图中左侧为 UIKit，右侧为 {Bee}:



Q1）{Bee} 提供了哪些控件？

参考代码：

1. framework/application/mvc/view/dom-element
2. framework/application/mvc/view/dom-element-ext

常用的控件有：

1. BeeUIButton - 按钮
2. BeeUIImageView - 图片
3. BeeUILabel - 标签
4. BeeUIScrollView - 多行多列的滚动视图
5. BeeUISwitch - ON/OFF
6. BeeUITextField - 单行输入框
7. BeeUITextView - 多行输入框

与 UIKit 原生控件相比，{Bee} 控件会更容易使用，体现在以下几点：

1. 支持 Signal 事件（后面介绍）
2. 支持 XML 模版（详见相关技术手册）
3. 支持 CSS 样式（详见相关技术手册）
3. 简单的生命期管理方法，如：

```
BeeUIButton * button = [BeeUIButton spawn];
```

@implementation MyButton

```
- (void)load
{
     ... initialize your button ...;
}

- (void)unload
{
     ... finalize your button ...;
}
```

```
@end
```

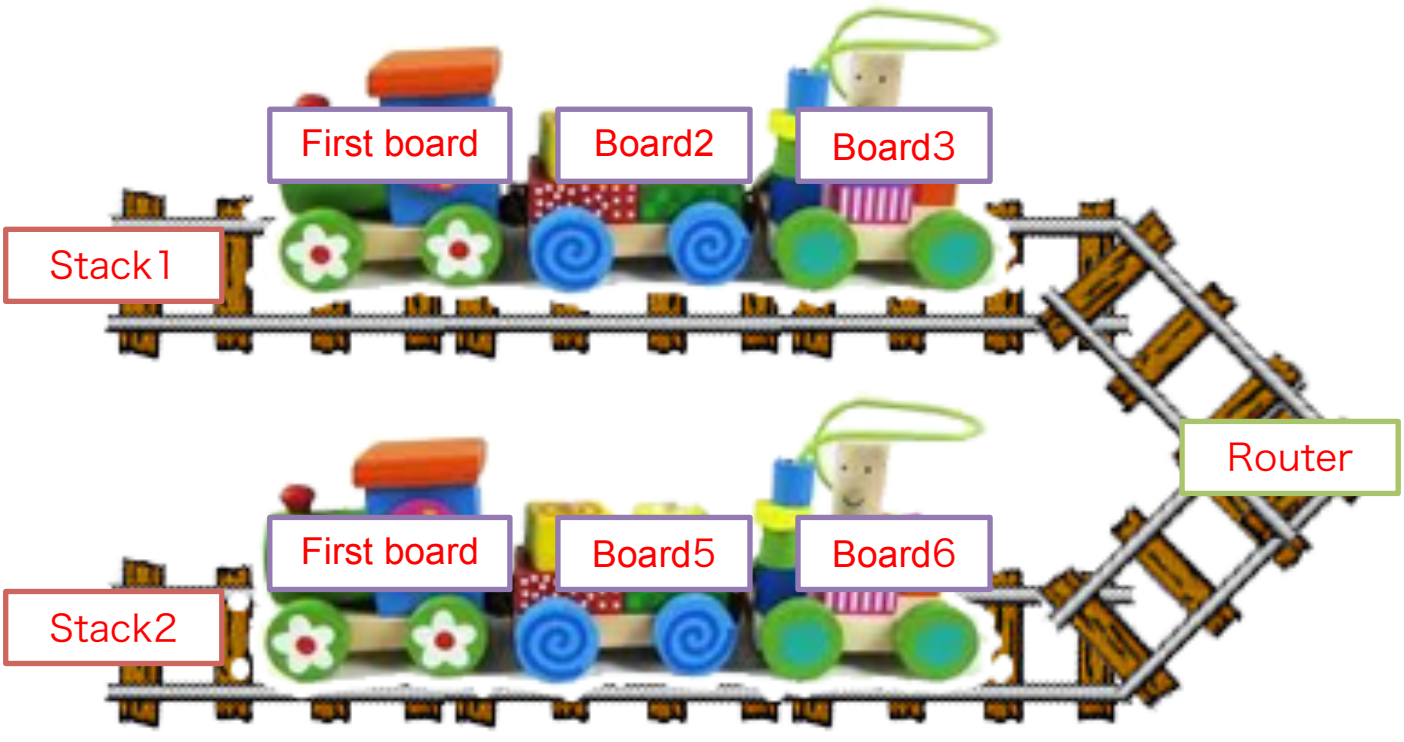
Q2）{Bee} 提供了哪些容器？

参考代码：

1. example/view_iPhone/AppBoard_iPhone.h
2. example/view_iPhone/Appboard_iPhone.m
3. framework/application/mvc/view/container

常用的容器有：
1. BeeUIBoard - 白板，继承自 UIViewController
2. BeeUIStack - 堆栈，继承自 UINavigationController
3. BeeUIRouter - 路由，继承自 UIViewController，管理着 Stack 的生命期及 Stack 之间切换显示

下图形象的描述了三种容器之间的关系：



通过以下代码，可以将容器可以做为 Skeleton 的界面入口，如：

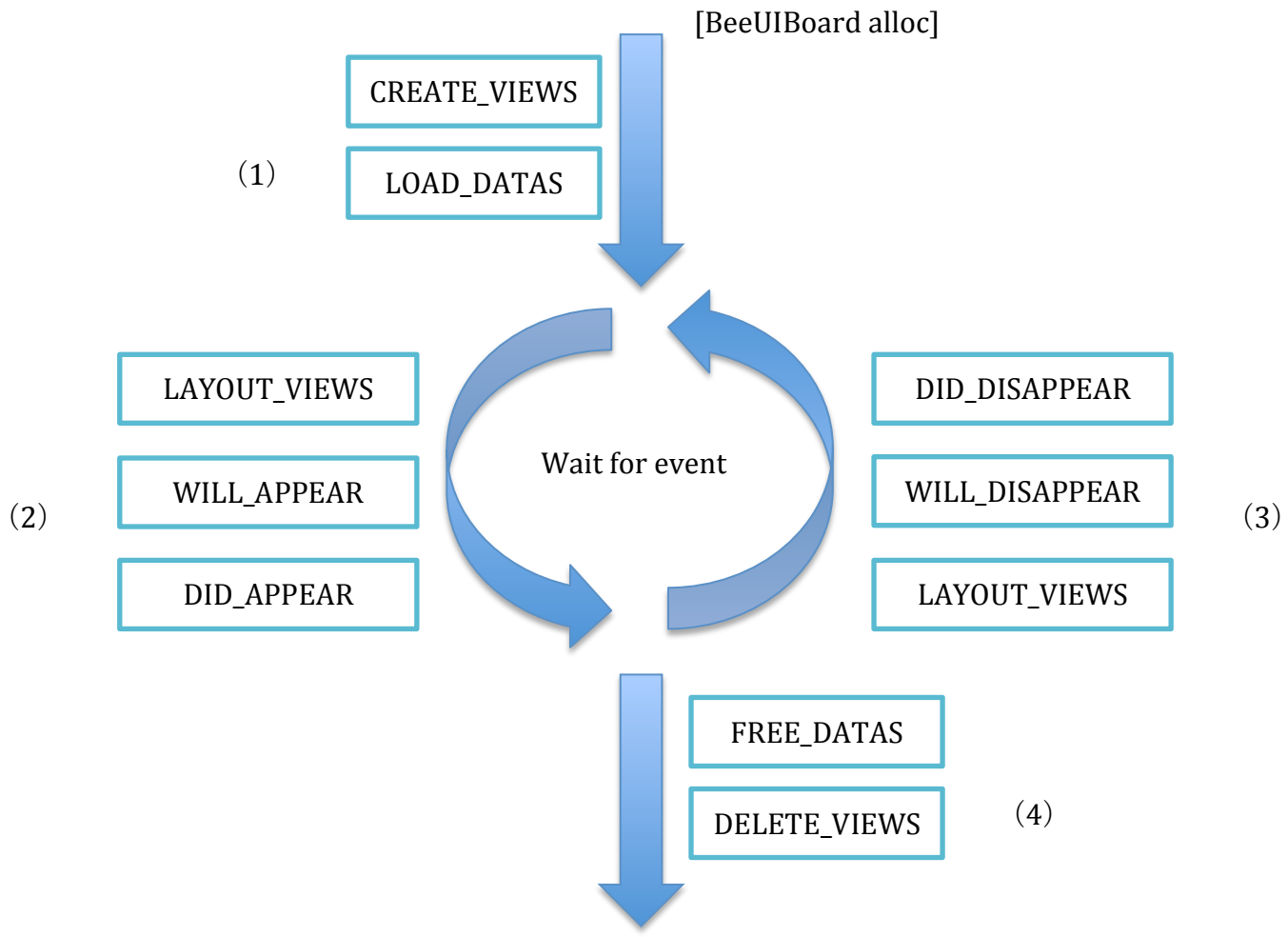
```
self.window.rootViewController = [BeeUIRouter sharedInstance];
self.window.rootViewController = [BeeUIStack ...];
self.window.rootViewController = [BeeUIBoard ...];
```

相比 UIKit 原生容器，{Bee} 容器使用简单，特性丰富：

1. 简单的生命期控制方法（ load/unload ）
2. 支持 XML template（后面介绍）
3. 支持 Signal 事件（后面介绍）

{Bee} Board 的生命期管理

BeeUIBoard 对 UIViewController 流程重新梳理如下：



CREATE_VIEWS - 创建视图，用户应创建自己的控件
LOAD_DATAS - 加载数据，用户应加载本地数据
LAYOUT_VIEWS - 重新布局，用户应按屏幕方向调整坐标
WILL_APPEAR - 将来可见，此时用户才可以触控作该视图
DID_APPEAR - 已经可见
WILL_DISAPPEAR - 将要消失，此时用户将无法触控作该视图
DID_DISAPPEAR - 已经消失
FREE_DATAS - 释放数据，用户应释放视图相关数据
DELETE_VIEWS - 删除视图，用户应删除内部的控件

{Bee} 提供简单的 Board 使用方式

```
@interface MyBoard : BeeUIBoard
@end

@implementation MyBoard


- (void)load
{
    // 初始化 BOARD 对象
}

- (void)unload
{
    // 释放 BOARD 对象
}


ON_SIGNAL3( BeeUIBoard, CREATE_VIEWS, signal )
{
    // TODO: 用户应创建自己的控件
}

ON_SIGNAL3( BeeUIBoard, DELETE_VIEWS, signal )
{
    // TODO: 用户应删除内部的控件
}

...
```

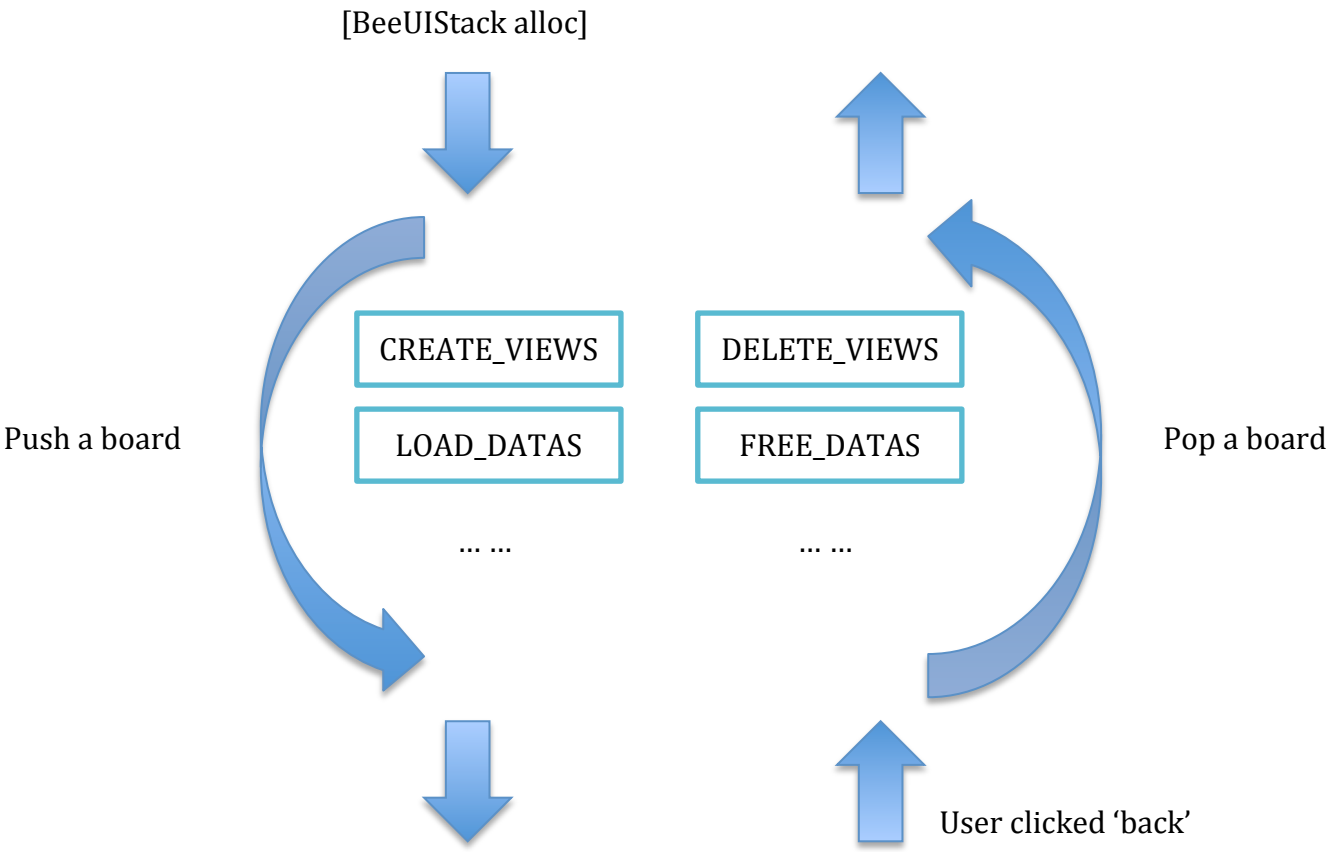
```
@end
```

{Bee} 提供简单的 Board 创建方式

```
MyBoard * board = [MyBoard board];
MyBoard * board = [[[MyBoard alloc] init] autorelease];
```

{Bee} Stack 的生命期管理

管理着 Board 的堆栈，即 Push 和 Pop:



当 Push 时，Board 被创建，CREATE_VIEWS 被触发。
在 Pop 时，Board 被销毁，DELETE_VIEWS 等被触发。

{Bee} 提供简单的 Stack 使用方式

```
[someStack pushBoard:… animated:YES];
[someStack popBoardAnimated:YES];
```

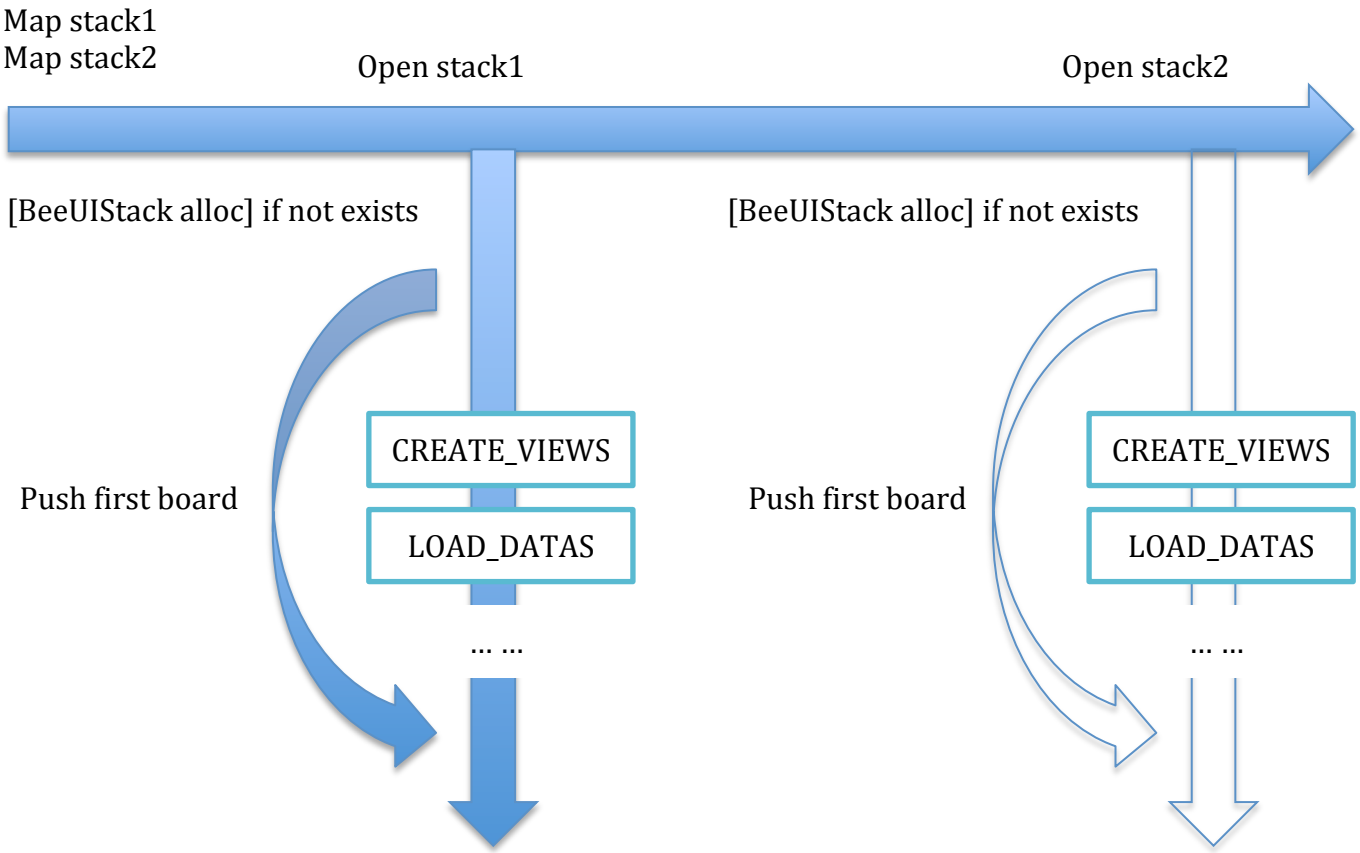
```
[someStack popToBoard:… animated:YES];
[someStack popToFirstBoardAnimated:YES];
```

{Bee} 提供简单的 Stack 创建方式

```
[BeeUIStack stack];
[BeeUIStack stackWithFirstBoardClass:[MyBoard class]];
[BeeUIStack stackWithFirstBoard:board];

[[BeeUIStack alloc] initWithName: … andFirstBoardClass: …];
[[BeeUIStack alloc] initWithName: … andFirstBoard: …];
```

{Bee} Router 的生命期管理

管理着多个 Stack 的显示和切换，即 Map 和 Open：



Router 有个非常有意思的特性“Lazy load”，即当 Router 有需求切换显示到一个 Stack 时，才去创建它。这样的好处是，往往 Router 被当做 Apps 的默认入口，如果在 Apps 启动时创建和加载了太多的

Stack 和 Board，势必会影响到启动性能，甚至卡死很久。按需分配后，这样的情况几乎很少出现了。

{Bee} 提供简单的 Router 使用方法

```
BeeUIRouter * router = [BeeUIRouter sharedInstance];

[router map:@”stack1” toClass:[MyBoard class]];
[router map:@”stack2” toBoard:[MyBoard board]];
[router open:@”stack1”];
```

这里，BeeUIRouter 默认为单件，不需要创建。
更多的示例，请参考 DEMO 工程。

Q3）{Bee} 如何编写界面布局？

记得有人说过这样的一个事实：

“移动开发者的 80%时间都花在了 UI 编码上”

我们引入了 XML 模版技术（XML template）解决该问题。

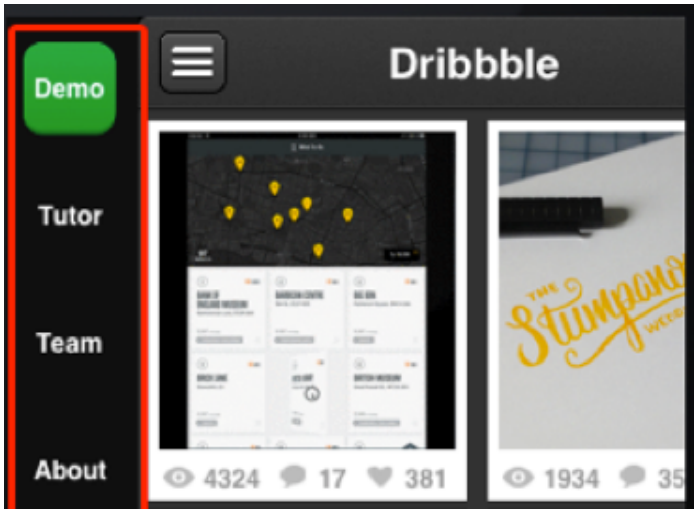
简单的讲，就是用 XML 语言描述出一个界面布局，在用户给定宽或高的情况下，能够按照规则计算出控件的坐标并显示出来。

我们来看一段代码：

```
<ui namespace="MenuBoard_iPhone">
    <linear orientation="v" class="wrapper">
      <image id="item-bg" class="item-icon"/>
      <linear orientation="v" class="item-wrapper">
            <label class="item-text">Demo</label>
            <button id="demo" class="item-mask"/>
      </linear>
      <linear orientation="v" class="item-wrapper">
            <label class="item-text">Tutor</label>
            <button id="tutor" class="item-mask"/>
      </linear>
```

```
            <linear orientation="v" class="item-wrapper">
                  <label class="item-text">Team</label>
                  <button id="team" class="item-mask"/>
            </linear>
            <linear orientation="v" class="item-wrapper">
                  <label class="item-text">About</label>
                  <button id="about" class="item-mask"/>
            </linear>
          </linear>
      </ui>
```

以上代码实现了 DEMO 工程中的简单菜单，效果如下：



{Bee} 提供简单的 XML 模版使用方法

```
@implementation MenuBoard_iPhone

...

SUPPORT_AUTOMATIC_LAYOUT( YES );
SUPPORT_RESOURCE_LOADING( YES );

...

@end
```

通过引用以下两个宏，{Bee}自动完成 XML 的加载和计算坐标：

1. SUPPORT_AUTOMATIC_LAYOUT(YES) - 是否自动布局
2. SUPPORT_RESOURCE_LOADING(YES) - 是否自动加载 xml

XML 加载的时机与条件

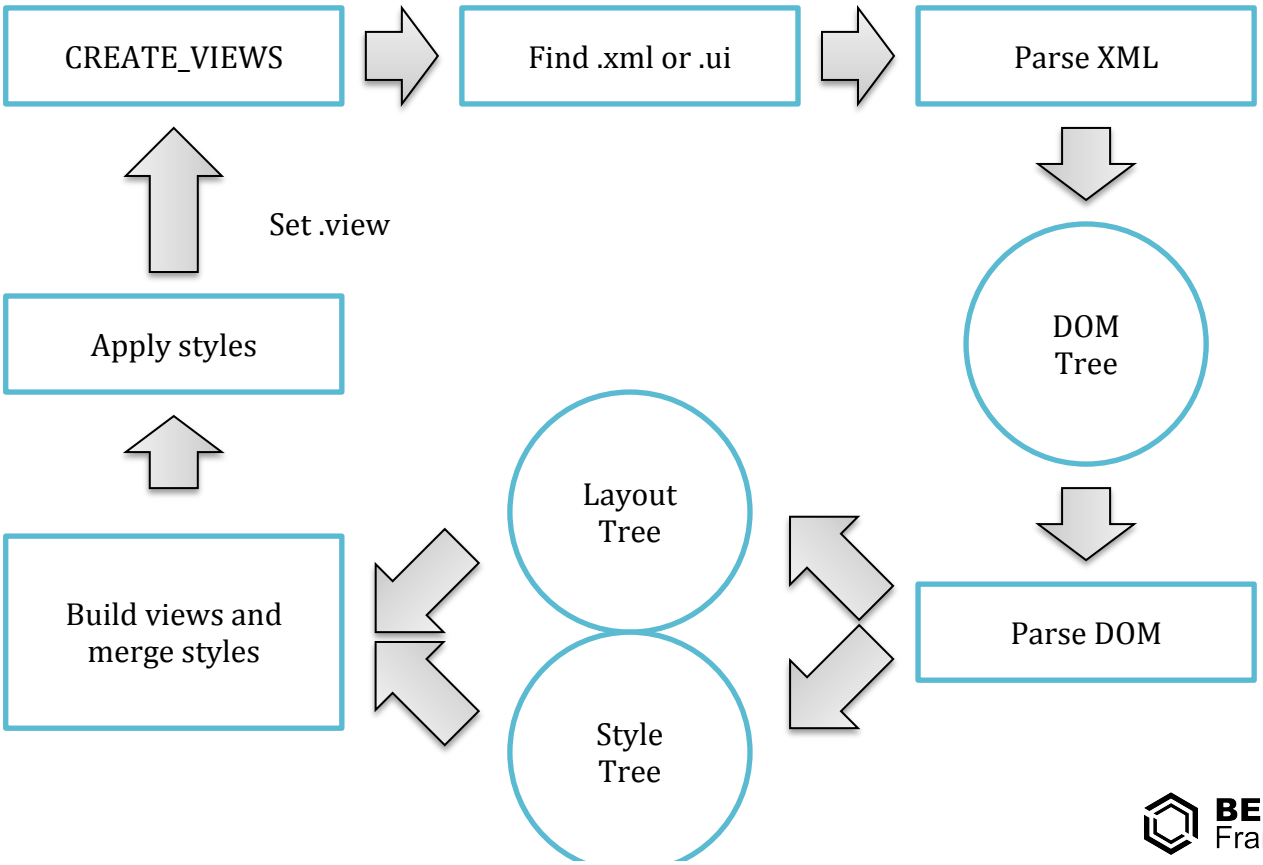
1. BeeUIBoard.CREATE_VIEWS，加载 XML
2. BeeUIBoard.LAYOUT_VIEWS，触发自动布局
3. BeeUIBoard.WILL_APPEAR，触发自动布局

XML 的手动加载及布局

```
self.FROM_RESOURCE( @"XXX.xml");
self.RELAYOUT();

someBoard.FROM_RESOURCE( @"YYY.xml");
someBoard.RELAYOUT();

someView.FROM_RESOURCE( @"YYY.xml");
someView.RELAYOUT();
```

通过该技术，开发者的工作流程得到了改进，将会节省大量的时间。

在 {Bee} 的内部，具体做了哪些事？



1. 加载 XML 资源
2. 解析 XML 为 DOM 树
3. 解析 DOM 树为 Layout 树及 Style 树
4. 根据 Layout 树，创建出 UIView 树
5. 根据 Style 树，合出样式，并赋值给 View 结点
6. 完成，并将 UIView 树的 Root 结点添加到 BeeUIBoard.view

其内部实现机制稍有些复杂，但对于一般开发者来讲并不必关心这些细节，学会怎样编写 XML 即可完成日常开发工作。

如果您对 XML Templates 技术有更多的兴趣，请参考《Bee templates 技术规范》来进一步了解。

Q4）{Bee} 如何改变界面样式？

我们另外提供了 CSS 样式技术，来进一步帮助开发者描述出界面细节，如：颜色，宽高，偏移，字体，背景图 等。

一般将 CSS 写在 XML 文件尾部，请参见 MenuBoard_iPhone.xml:

```
<ui namespace="MenuBoard_iPhone">

    ...
    <style type="text/css">
      .wrapper {
            width: 60px;
            height: 100%;
      }
      .item-wrapper {
            width: 100%;
            height: 54px;
            margin-top: 6px;
      }
      .item-icon {
            width: 100%;
            height: 54px;
            position: absolute;
            left: 0px;
            top: 6px;
            image-mode: center;
```

```
                image-src: url(menu-icon-green.png);
            }


        </style>
        ...

    </ui>
```

请参考《Bee templates 技术规范》来进一步了解。

Q5）{Bee} 如何填写界面数据？

这里介绍一项新技术，{Bee} UIQuery，一项高效的界面元素增删改查技术。如果您对 jQuery 非常熟悉，那么这项技术将进一步加速您的开发节奏。不熟悉也没关系，简单了解语法即可上手。

来看个例子，修改 MenuBoard_iPhone 中 id 为”tutor”的按钮的文字为“test”：

```
@implementation MenuBoard_iPhone
{
    $(@”tutor”).DATA( @”test” );
}
@end
```

值得注意的是，$操作符是上下文相关的，即在不同的方法中操作的是不同的对象，在上面的代码中，$操作是 MenuBoard_iPhone.view。

简单分析其原理为：

1. 根据当前上下文环境，找到 root view
2. 分析$()中的表达式，解析出 tagString
3. 查找 root view 中 tagString 相等的一个或多个 UIView
4. 返回 UIView 的一个集合对象 UICollection
5. 通过对 UICollection 的点操作符，可以直接操作 View 的增删改查

常用的 UI query 语句有：

```
$(@"xxx").SHOW();                  - 显示
$(@"xxx").HIDE();                  - 隐藏
$(@"xxx").FIND( @"yyy" );          - 查找子 view
$(@"xxx").DATA( @"zzz" );          - 赋值
$(@"xxx").FOCUS();                 - 激活（输入框）
$(@"xxx").BLUR();                  - 取消（输入框）
$(@"xxx").ENABLE();                - 使之可用
$(@"xxx").DISABLE();               - 使之不可用
$(@"*").EACH();                    - 遍历每个元素
```

绝大部分语法与 jQuery 相似，也可以参考 jQuery 官网的文档完成学习。

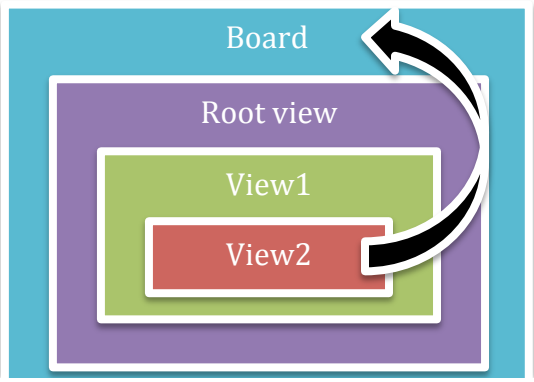
Q6）{Bee} 如何响应界面事件？

这里我们主要介绍{Bee} Signal 技术，及其简单的使用方法。

您可能会问，为什么不用 Delegate?
本质上讲，Delegate 并不算是“事件”，Delegate 只是传递事件一种手段。传统的 Delegate 有哪些缺点？或者说，理想中的“事件”应具备哪些特性？我觉得：

1. 事件应该对象化，而不是简单函数回调。
2. 事件应该具有名字，让开发者方便判断。
3. 事件应该存在于上下文环境中，让开发者可以获取参数数据。
4. 事件应该支持“冒泡”传递方式。
5. 事件与它的接收者应该是松耦合的。

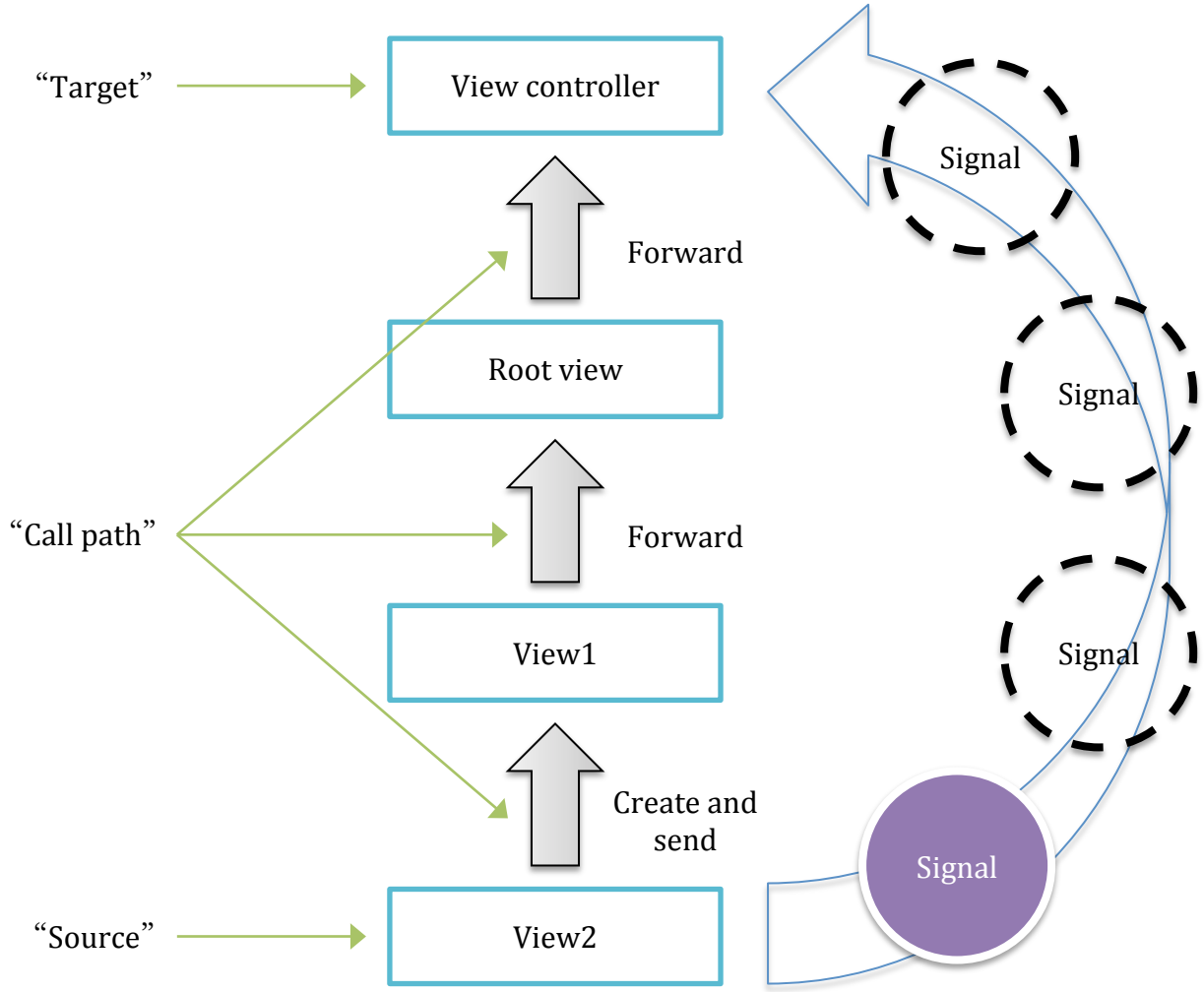
试想一下，实际开发中有没有遇过这样的场景？



怎样将 View2 的事件逐层传递给 Board? 方法有很多种，例如:

1. 使用 Delegate
2. 使用 Notification
3. 使用 Responder chain

对比以上方法，我们来看一下 Signal 如何实现的?



Signal 为 UIView 及 UIViewController 提供了事件的路由机制:

1. Signal 从 Source 发出
2. 沿 Call path 逐层传递，即 View2 -> ⋯ -> View controller

3. 由 Target 接收处理
4. 途中每一站可选择“处理”或“忽略”，并向上层“转发”

事件实体为“Signal”，即 BeeUISignal 类对象
事件发起者称为“Source”
事件接收者称为“Target”，一个 UIViewController 或顶层 UIView
整个传递路径称为“Call path”

Signal 怎样定义?

Signal 的命名必须遵守如下规范:

从以上 Signal 全名可以看出，是表示 View2 被点击的事件。可以看出，遵守 Signal 命名规范是有助于开发者理解事件的语义。

“signal” - 固定前辍
“View2” - 命名空间，通常是类名
“TAPPED” - 事件名称，通常开发者自定义

为了简化书写， 提供了以下三个宏:

AS_SIGNAL(name) - 声明 Signal
DEF_SIGNAL(name) - 定义 Signal
DEF_SIGNAL_ALIAS(name, alias) - 定义 Signal 别名

使用方法如下:

```
@interface View2
AS_SIGNAL( TAPPED )
```

```
@end

@implementation View2
DEF_SIGNAL( TAPPED )
@end
```

Signal 有哪些特点?

1. 类名做为全名空间， 防止全局名字污染
2. 子类可继承
3. 子类可重写
4. 书写简单，开发者易懂

Signal 怎么引用?

1. 如果在类方法中，引用当前类的 Signal:

```
+ (void)method
{
    ...    self.TAPPED    ...;
}
```

2. 如果在类方法中，引用其他类的 Signal:

```
+ (void)method
{
    ...    View2.TAPPED    ...;
}
```

3. 如果在实例方法中，引用当前对象类的 Signal:

```
- (void)method
{
    ...    self.TAPPED    ...;
}
```

Signal 怎样判断? 和获取用户数据?

```
ON_SIGNAL(signal )
```

```
	{
		if ( [signal is:View2.TAPPED] )
		{
			NSObject * userObject = signal.object;

			// do something with userObject
		}
	}
```

Signal 怎么发送？和附带参数？

```
- (void)method
{
	[self sendUISignal:self.TAPPED];
	[self sendUISignal:self.TAPPED withObject:XXX];
}
```

Signal 怎么接收？和转发？

默认的 signal 处理方法为 handleUISignal，开发者重载实现即可：

```
	- (void)handleUISignal:(BeeUISignal *)signal
	{
		[super handleUISignal:signal];

		if ( [signal is:View2.TAPPED] )
		{
		}
	}
```

{Bee} 也提供了用于 signal 过滤的特殊命名办法，即：

```
	- (void)handleUISignal_<Name space>
	- (void)handleUISignal_<Name space>_<Event name>
	- (void)handleUISignal_View2
	- (void)handleUISignal_View2_TAPPED
```

为了减少手写代码量，我们另外提供了 3 个宏，具有同等效果即：

1. 默认处理方法

```
	ON_SIGNAL( signal )
```

```
{
}
```

2. 过滤处理所有来自 UIView2 类对象的 Signal

```
ON_SIGNAL2( UIView2, signal )
{
}
```

3. 过滤处理所有来自 UIView2 类对象的 TAPPED 事件 Signal

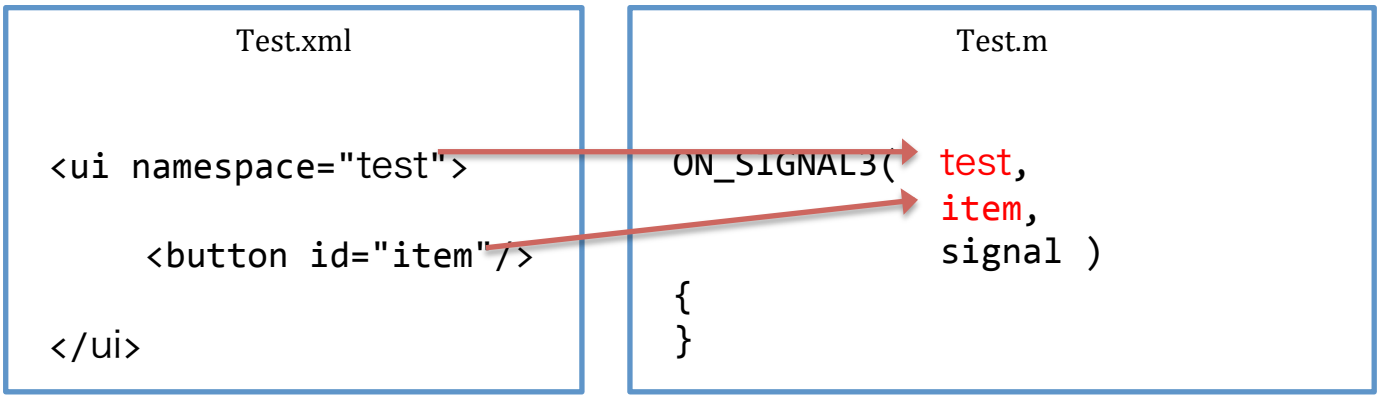
```
ON_SIGNAL3( UIView2, TAPPED, signal )
{
}
```

如果在同一类中，添加了以上三种处理方法，将按以下优先级调用：

```
ON_SIGNAL3 -> ON_SIGNAL2 -> ON_SIGNAL
```

Signal binding 与 XML template?

Signal 技术同样可以应用于 XML 模版中去，类似于 WebView 的 JS 绑定的技术，我们把它称为 Signal binding，对应关系如下：



即，使用 ON_SIGNAL3 宏来粘合 XML 与 Signal 对应关系，第一个参数为 xml namespace，第二个参数为 id，第三个参数为 signal 本身。

请参考我们提供的 DEMO 工程来进一步学习。

深入学习 - 理解 Controller

参考源码:

Framework/application/mvc/controller

很多开发者认为，在 App 中 Controller 就是 UIViewController，负责处理用户输入，调用网络接口，存取数据库，更新显示。

理论上讲，视图应该与逻辑分离，应该由 View 接受用户输入，再调用 Controller 来完成用户的需求。Controller 应该只做两件事:

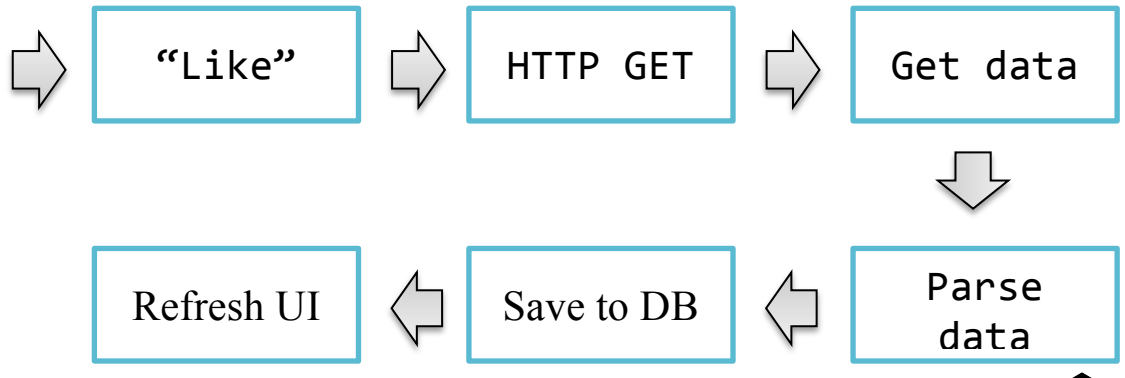
1. 执行逻辑
2. 调用 Model 存储

想想，一个 App 可以适配 iPhone/iPad 两种分辨率，但逻辑是可以抽象出来被复用的。

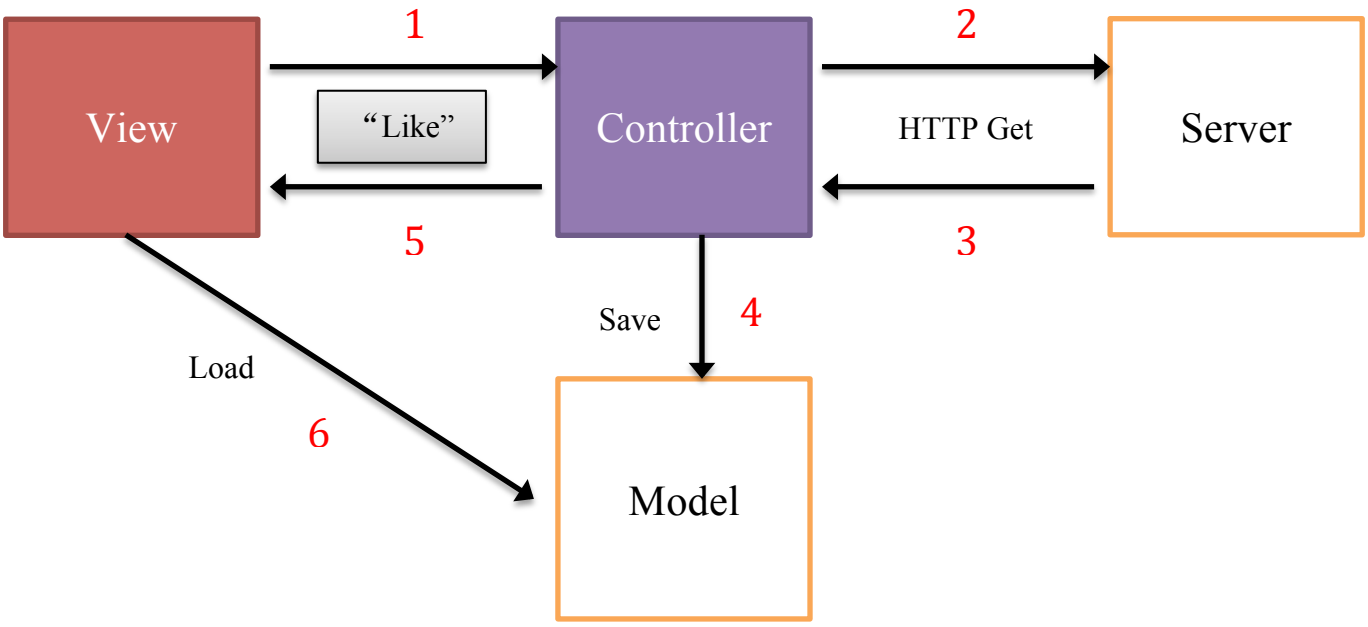
常见的逻辑有很多种，如:

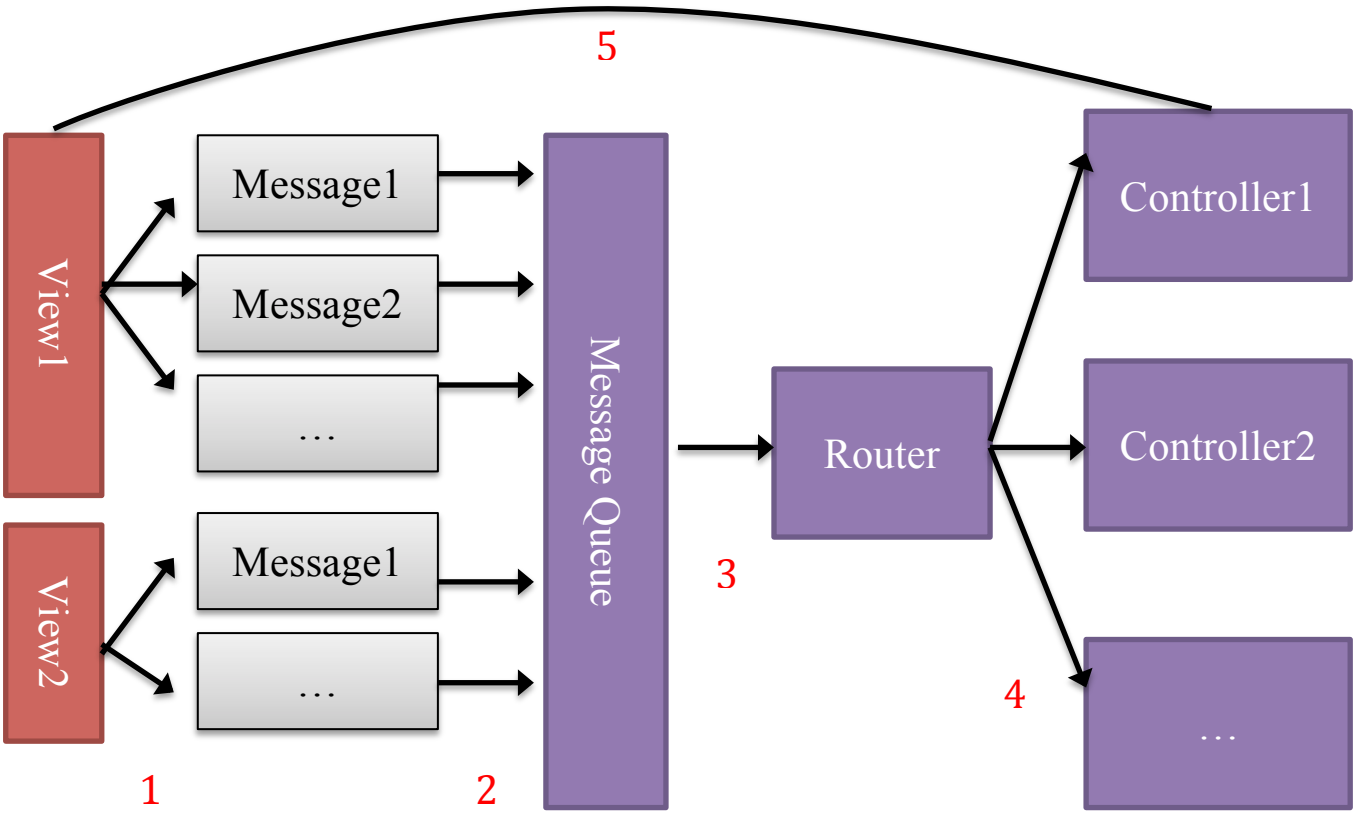
1. 按下按钮来“喜欢”一张照片
2. 按下按钮来“修改”个人资料
3. 下拉刷新来“获取”文章列表

简单分解一下流程:



类似的逻辑都大同小异，我们来看看在 {Bee} 是怎样实现的?





Message: 消息实体。
Message Queue: 消息队列，保证消息有序的执行；
Router: 路由器，转发消息达到正确的 Controller；
Controller: 控制器，消息的执行者；

消息有以下特性：
1. 消息是有名字的，通过命名与 Controller 关联
2. 消息是有状态的
3. 消息本身没有逻辑，但保存有逻辑执行的上下文
4. 消息是异步执行的，可以被取消
5. 消息是可以并发的
6. 消息可以设置超时
7. 消息是可选回调的

相关的两个类：
1. BeeMessage - Message 的实体类
3. BeeController - Controller 的实体类

Controller 如何定义？

```
@interface MyController : BeeController
@end

@implementation MyController
@end
```

Controller 何时加载？

当 Router 路由时，被自动创建和加载，不需要手动创建和加载。

Message 如何定义？

```
@interface MyController
AS_MESSAGE( LIKE )
@end
```

```
@implementation MyController
DEF_MESSAGE( LIKE )
@end
```

Message 如何引用?

MyController.LIKE，即 <类名>.<消息名>

Message 如何发送? 和附带参数?

```
self.MSG( MyController.LIKE );
self.MSG( MyController.LIKE ).INPUT( @"key", @"value" );
```

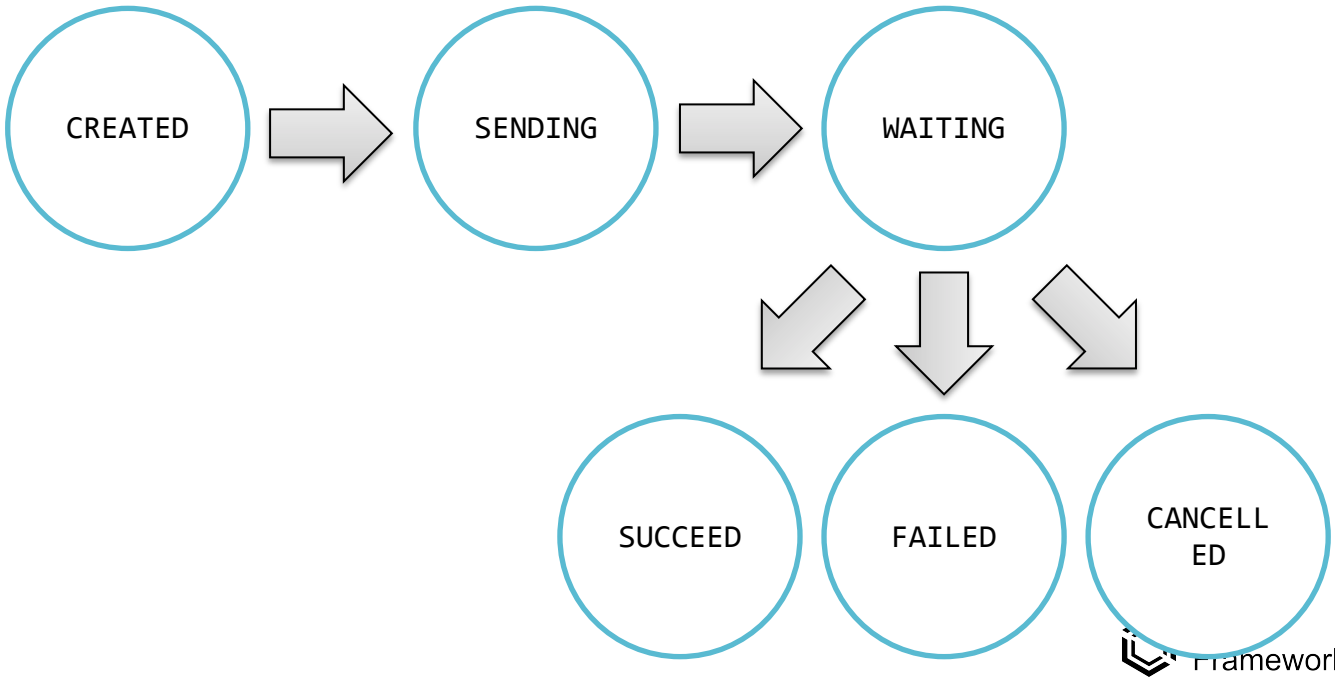
Message 如何被路由?

```
@implementation MyController
- (void)LIKE:(BeeMessage *)msg
{
}
@end
```

通过在 Controller 中定义与 Message 相同名字的方法来完成路由。

Message 如何被执行?

一般消息的执行会经由起始状态到终结状态的几个转换过程:



STATE_CREATED	- （起始）等待发送
STATE_SENDING	- 等待执行
STATE_WAITING	- 等待完成
STATE_SUCCEED	- （终结）执行成功
STATE_FAILED	- （终结）执行失败
STATE_CANCELLED	- （终结）用户取消

根据实际需求，何时算作成功或失败是由开发者来控制的，{Bee} 提供了一些简单的控制开关：

```
message.succeed       = YES/NO;
message.failed        = YES/NO;
```

或

```
message.state         = BeeMessage.STATE_XXX;
```

即可用做状态判断 Getter，也可用做状态改变 Setter，例如：

```
@implementation MyController

- (void)LIKE:(BeeMessage *)msg
{
      if ( msg.sending )
      {
            // do something when sending
            msg.succeed = YES;
            return;
      }
      else if ( msg.succeed )
      {
            // do something when succeed
      }
}

@end
```

Message 如何响应？

在 {Bee} 中，我们要求消息的响应者应实现以下方法：

```
@implementation MyBoard

- (void)handleMessage: (BeeMessage *)msg
{
      if ( msg.sending )
      {
      }
      else if ( msg.succeed )
      {
      }
      else if ( msg.failed )
      {
      }
      else if ( msg.cancelled )
      {
      }
}
```

每当状态转换时，Router 会通过 Message 中保存的 responder 回调。可以注意到一个细节，这样设计好处是让 View 与 Controller 代码结构一致性。

另外 {Bee} 也提供了以下几个宏，简化编码：

```
ON_MESSAGE( msg )
ON_MESSAGE2( MyController, msg )
ON_MESSAGE3( MyController, LIKE, msg )
```

在后面的章节，我们将介绍 Controller 代码生成技术 Scaffold，如果有兴趣请前往查看。

深入学习 - 理解 Model

参考源码:

Framework/application/mvc/model

{Bee} Model 本身实现比较简单，但相关联的组件众多，具体使用方法建议参考 DEMO 工程:

1. 数据对象化、序列化与反序列化

    参考代码
    /framework/system/foundation
    /framework/system/database

    BeeActiveObject - 支持对象到字符串的序列化与反序列化
    BeeActiveRecord - 支持对象到 SQLITE 的存储

2. 数据缓存

    参考代码 /framework/system/cache

    BeeFileCache - 文件缓存
    BeeMemoryCache - 内存缓存
    BeeKeychain - 加密钥匙存
    BeeUserDefaults - 用户 Plist 数据

高级编程 - Core Cache

参考源码:

Framework/system/cache

1. 钥匙串 Keychain

```
{
    NSString * value = [self keychainRead:@”key”];
    [self keychainWrite:value forKey:@”key”];
    [self keychainDelete:@”key”];}
}
```

2. 用户数据 UserDefaults

```
{
    NSString * value = [self userDefaultsRead:@”key”];
    [self userDefaultsWrite:value forKey:@”key”];
    [self userDefaultsDelete:@”key”];}
}
```

3. 内存缓存 Memory cache

```
{
    [[BeeMemoryCache sharedInstance] objectForKey:@”key”];
    [[BeeMemoryCache sharedInstance] setObject:obj forKey:@”key”];
    [[BeeMemoryCache sharedInstance] removeObjectForKey:@”key”];
    [[BeeMemoryCache sharedInstance] removeAllObjects];
}
```

4. 文件缓存 File cache

```
{
    [[BeeFileCache sharedInstance] objectForKey:@”key”];
    [[BeeFileCache sharedInstance] setObject:obj forKey:@”key”];
    [[BeeFileCache sharedInstance] removeObjectForKey:@”key”];
    [[BeeFileCache sharedInstance] removeAllObjects];
}
```

高级编程 - Core Database

参考源码:

Framework/system/database

1. 打开数据库

```
[BeeDatabase openSharedDatabase:@"test"];
```

2. 关闭数据库

```
[BeeDatabase closeSharedDatabase];
```

3. 建表

```
self
    .DB
    .TABLE( @"blogs" )
    .FIELD( @"id", @"INTEGER" ).PRIMARY_KEY().AUTO_INREMENT()
    .FIELD( @"type", @"TEXT" )
    .FIELD( @"date", @"TEXT" )
    .FIELD( @"content", @"TEXT" )
    .CREATE_IF_NOT_EXISTS();
```

4. 索引

```
self.DB.TABLE( @"blogs" ).INDEX_ON( @"id", nil );
```

5. 插入

```
self
.DB
.FROM( @"blogs" )
.SET( @"type", @"Test" )
.SET( @"date", [[NSDate date] description] )
.SET( @"content", @"Hello, world!" )
.INSERT(); // write once
```

6. 删除

```
self
.DB
.FROM( @"blogs" )
.WHERE( @"id", @1 )
.DELETE();
```

7. 计数

```
self
.DB
.FROM( @"blogs" )
.COUNT();
```

8. 查询

```
self
.DB
.FROM( @"blogs" )
.OFFSET( 0 )
.LIMIT( 10 )
.GET();
```

9. 更新

```
self
.DB
.FROM( @"blogs" )
.SET( @"content", @"Hello, world!" )
.WHERE( @"id", @1 )
.GET();
```

{Bee} 0.3.0 以后版本，不建议开发直接操作数据库，取而代之的使用 BeeActiveRecord 完成自动建表和存储操作。值得注意的是，AR 支持自动字段映射及类嵌套，可以完成复杂的关联存储，举个例子。

```
@interface Record1 : BeeActiveRecord
@property (nonatomic, retain) NSNumber *         lid;
@property (nonatomic, retain) NSNumber *         lat;
@property (nonatomic, retain) NSNumber *         lon;
@end
```

```
@interface Record2 : BeeActiveRecord
@property (nonatomic, retain) NSNumber *        uid;
@property (nonatomic, retain) NSString *        name;
@property (nonatomic, retain) Record1 *         location;
@end
```

1. 插入

```
Record1 * obj = [Record2 record];
obj.name = @”test”;
obj.SAVE();
```

2. 删除

```
Record1 * obj = Record2.DB.FIRST_RECORD();
If ( obj )
{
    obj.DELETE();
}
```

3. 查询

```
NSArray * array = Record2.DB.GET_RECORDS();
NSArray * array2 = Record2.DB.WHERE( @”uid”, @0 ).GET_RECORDS();
```

4. 更新

```
Record2 * obj = Record2.DB.FIRST_RECORD();
obj.name = @”test”;
obj.SAVE();
```

5. 创建

```
Record2 * obj = [Record1 record];
```

6. 复制

```
Record2 * obj = [Record2 record: another];
```

高级编程 - Core Foundation

参考源码:

Framework/system/foundation

1. 断言 Assertion

```
ASSERT( NO != result );
```

2. 分级日志 Log

```
INFO( @"format %d", 1234 );
PERF( @"format %d", 1234 );
WARN( @"format %d", 1234 );
ERROR( @"format %d", 1234 );

VAR_DUMP( obj );
OBJ_DUMP( obj );
```

3. 性能检测 Performance

```
PERF_ENTER_( tag )
{
    ...
}
PERF_LEAVE_( tag )

PERF_MARK( tag2 );
{
    ...
}
PERF_MARK( tag3 );

NSTimeInterval time = PERF_TIME( tag2, tag3 );
```

4. 运行时 Runtime

```
BREAK_POINT();
PRINT_CALLSTACK( 10 );
```

```
NSArray * stacks = [BeeRuntime callstacks:10];
NSArray * frames = [BeeRuntime callframes:10];
```

5. 单件 Singleton

```
@interface MyClass
AS_SINGLETON( MyClass )
@end

@implementation MyClass
DEF_SINGLETON( MyClass )
@end

MyClass * sharedObj = [MyClass sharedInstance];
```

6. 多线程 Thread

```
BACKGROUND_BEGIN
{
    // do something in sub thread

    FOREGROUND_BEGIN
    {
           // do something in main thread
    }
    FOREGROUND_COMMIT
}
BACKGROUND_COMMIT
```

7. 时钟 Ticker

```
[self observeTick];
[self unobserveTick];

ON_TICK( time )
{
    // do something
}
```

高级编程 - Core Network

参考源码:

Framework/system/network

1. 发送请求

```
self
.HTTP_GET( @”http://www.qq.com” )
.PARAM( @”key”, @”value” )
.PARAM( [NSDictionary dictionary] )
.TIMEOUT( 10 );


self
.HTTP_POST( @”http://www.qq.com” )
.PARAM( @”key”, @”value” )
.PARAM( [NSDictionary dictionary] )
.FILE( @”test.txt”, [NSData data] )
.FILE_ALIAS( @”test.txt”, [NSData data], @”name.txt” )
.TIMEOUT( 10 );
```

2. 取消请求

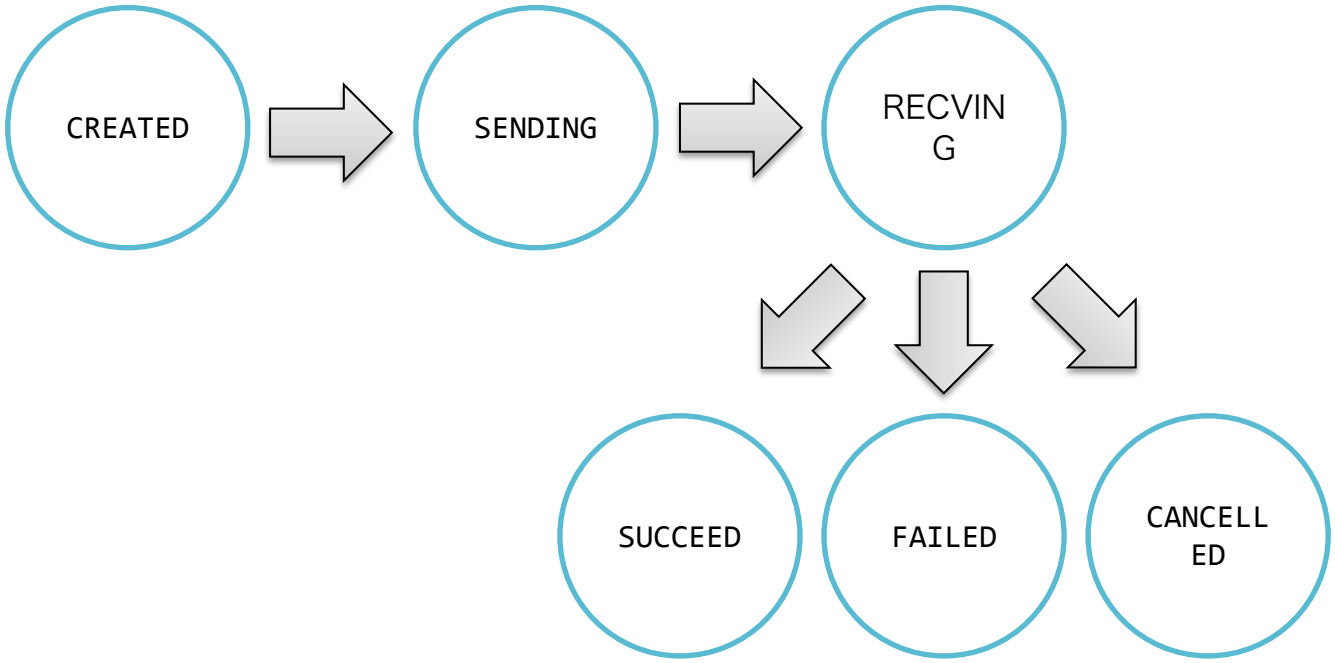
```
[self cancelRequests];
```

3. 处理响应

```
- (void)handleRequest:(BeeUIRequest *)req
{
    if ( req.sending )
    {
    }
    else if ( req.recving )
    {
    }
    else if ( req.failed )
    {
    }
    else if ( req.succeed )
    {
    }
```

```
    else if ( req.cancelled )
    {
    }
}
```

网络请求的状态轮转如下图：



每当状态轮转，都将回调给发起者，通过以下 Getter 判断：

request.sending	- （起始）正在发送
request.sendProgressed	- 发送进度更新
request.recving	- 正在接收
request.recvProgressed	- 接收进度更新
request.failed	- （终止）失败
request.succeed	- （终止）成功
request.cancelled	- （终止）用户取消

4. 获取进度

```
- (void)handleRequest:(BeeUIRequest *)req
{
    if ( req.sendProgressed )
    {
```

```
            CGFloat progress = req.HTTPUploadProgress;
        }
        else if ( req.recvProgressed )
        {
            CGFloat progress = req.HTTPDownloadProgress;
        }
    }
```

5. 获取数据

```
    - (void)handleRequest:(BeeUIRequest *)req
    {
        if ( req.succeed )
        {
            NSData *        data = req.response;
            NSArray *       array = req.responseJSONArray;
            NSDictionary *  dict = req.responseJSONDictionary;
        }
    }
```

6. 与 Message 结合

```
    - (void)handleMessage:(BeeMessage *)msg
    {
        if ( msg.sending )
        {
            self.HTTP_GET( … );
        }
        else if ( msg.succeed )
        {
            NSData *        data = msg.HTTPResponseData;
            NSString *      string = msg.HTTPResponseString;

            NSObject *      obj = msg.HTTPResponseJSON;
            NSDictionary *  dict = msg.HTTPResponseJSONDictionary;
            NSArray *       array = msg.HTTPResponseJSONArray;
        }
    }
```

高级编程 - Core Service

参考源码：

Framework/system/service

从 {Bee} 0.4.0 版本以后开始支持后台服务 Service 功能，特性如下：

1. 即装即用，0 配置
2. 自动加载，无需创建
3. 后台运行，无需界面

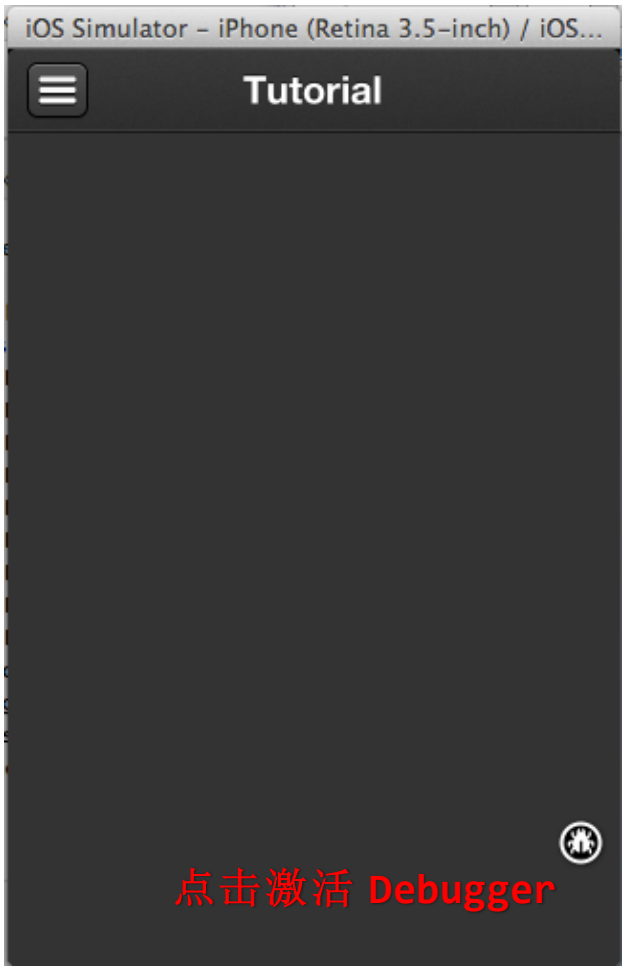
以常用的 ServiceDebugger 举例，当您需要 Debugger 服务时，仅需要/Services/ServiceDebugger 目录拖入工程，重新编译运行时，可以看到该服务已经在运行了。

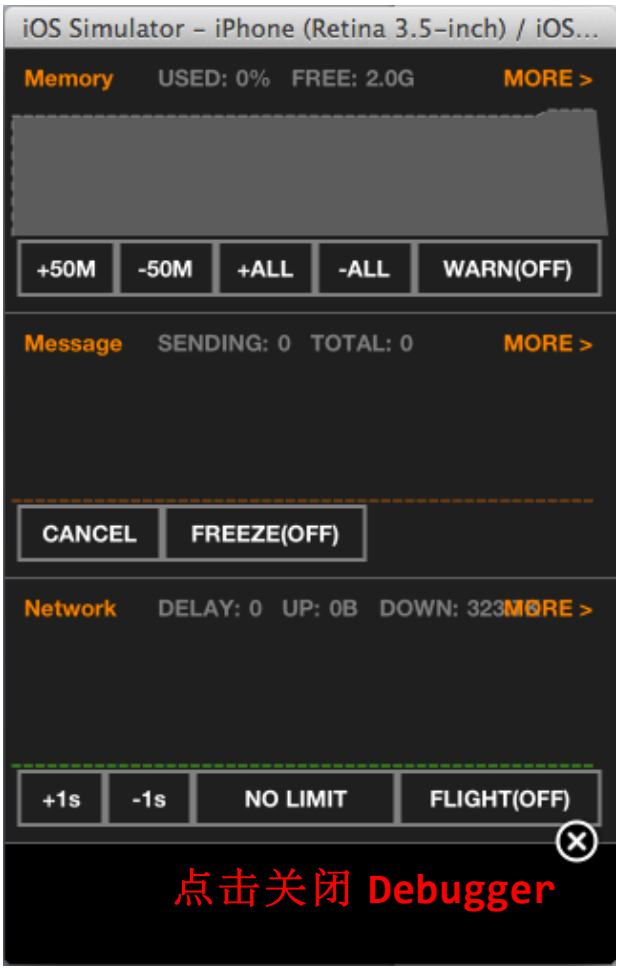
{Bee} 提供的常用服务有：

ServiceDebugger - 调试器服务
ServiceInspector - 视图观察服务
ServiceLocation - 位置服务
ServicePush - 推送服务

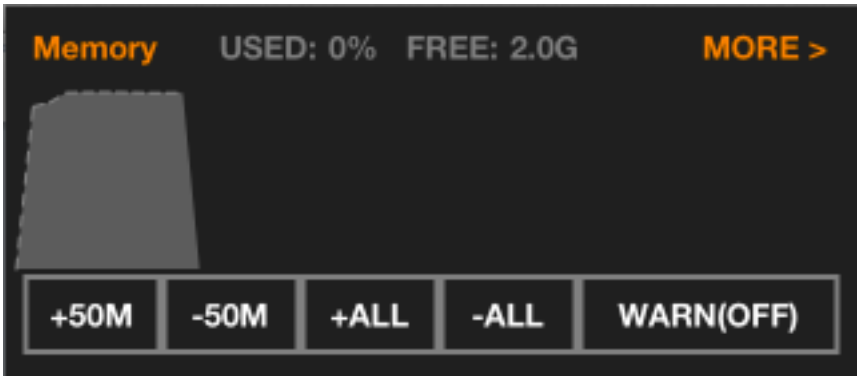
附加工具 - Debugger

启用方法：将 Services/ServiceDebugger 拖入工程，编译运行。

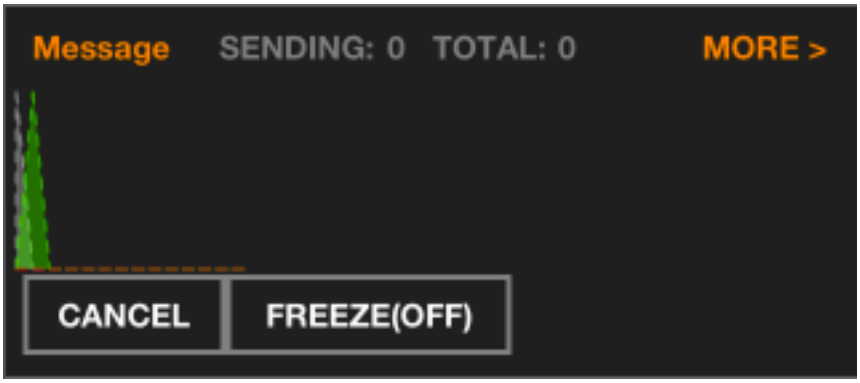




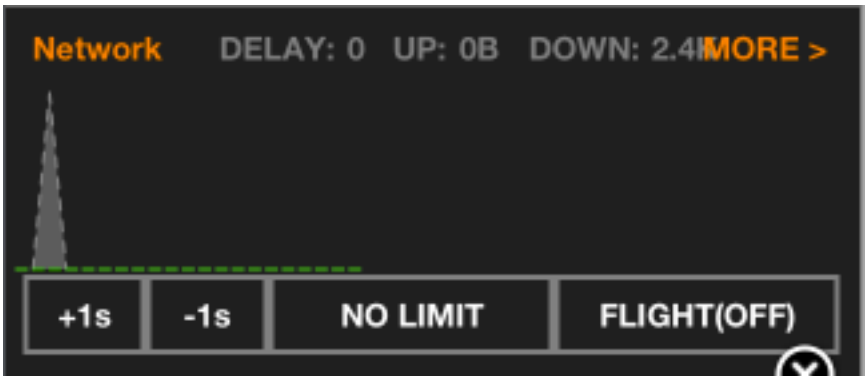
1. 内存状态，从左到右按钮功能依次为：
分配 50M 内存、释放 50M 内存，占满内存，释放内存，警告。



2. 消息状态，从左到右按钮功能依次为：
取消所有，冻结所有



3. 网络状态，从左到右按钮功能依次为：
所有请求延长 1 秒，所有请求缩短 1 秒，模拟 2G，飞行模式



附加工具 - Inspector

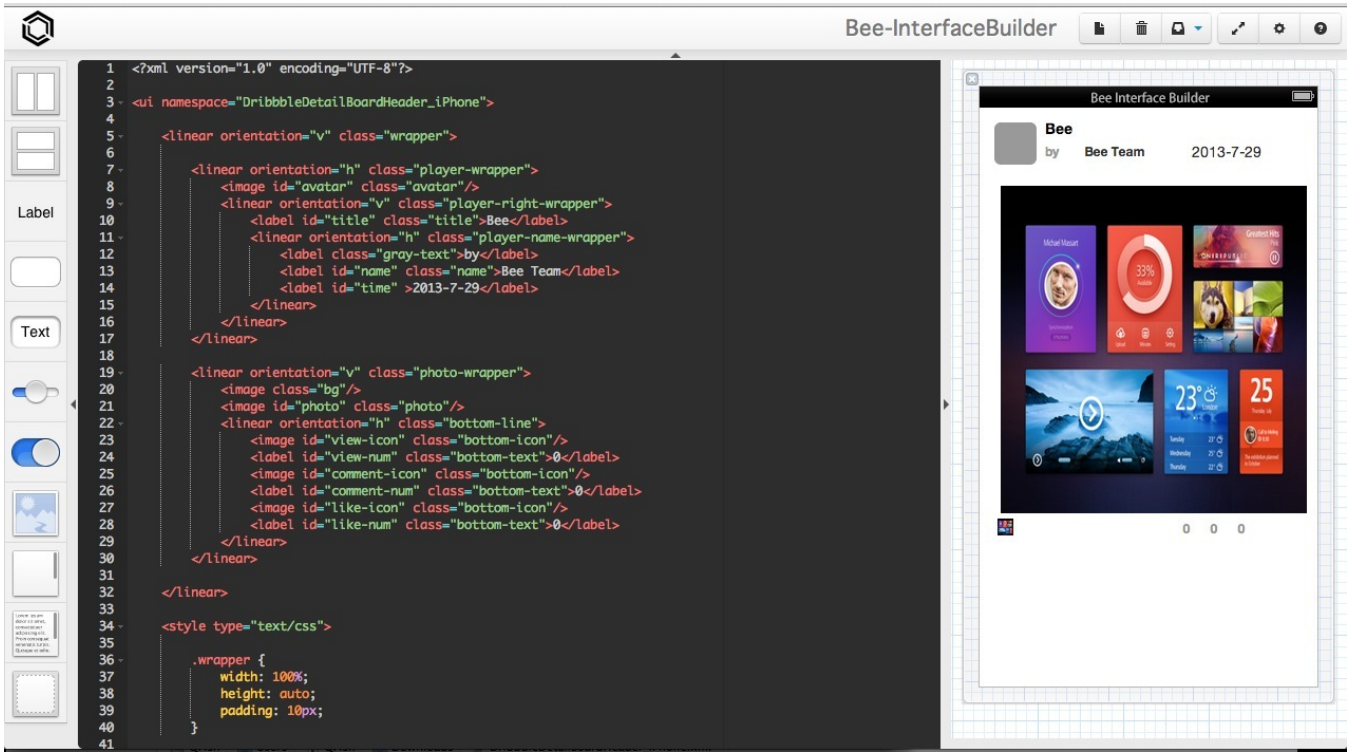
启用方法：将 Services/ServiceInspector 拖入工程，编译运行。

每次触击屏幕时，控制台也会输出相关 LOG 给出具体参数说明。

附加工具 - Interface builder

请访问 ib.bee-framework.com 查看，提供了以下功能：

1. 所见即所得
2. 常用控件和布局，一键生成代码
3. 智能语法提示，支持 ZenCoding
4. 线框图方式显示
5. 更多主题支持

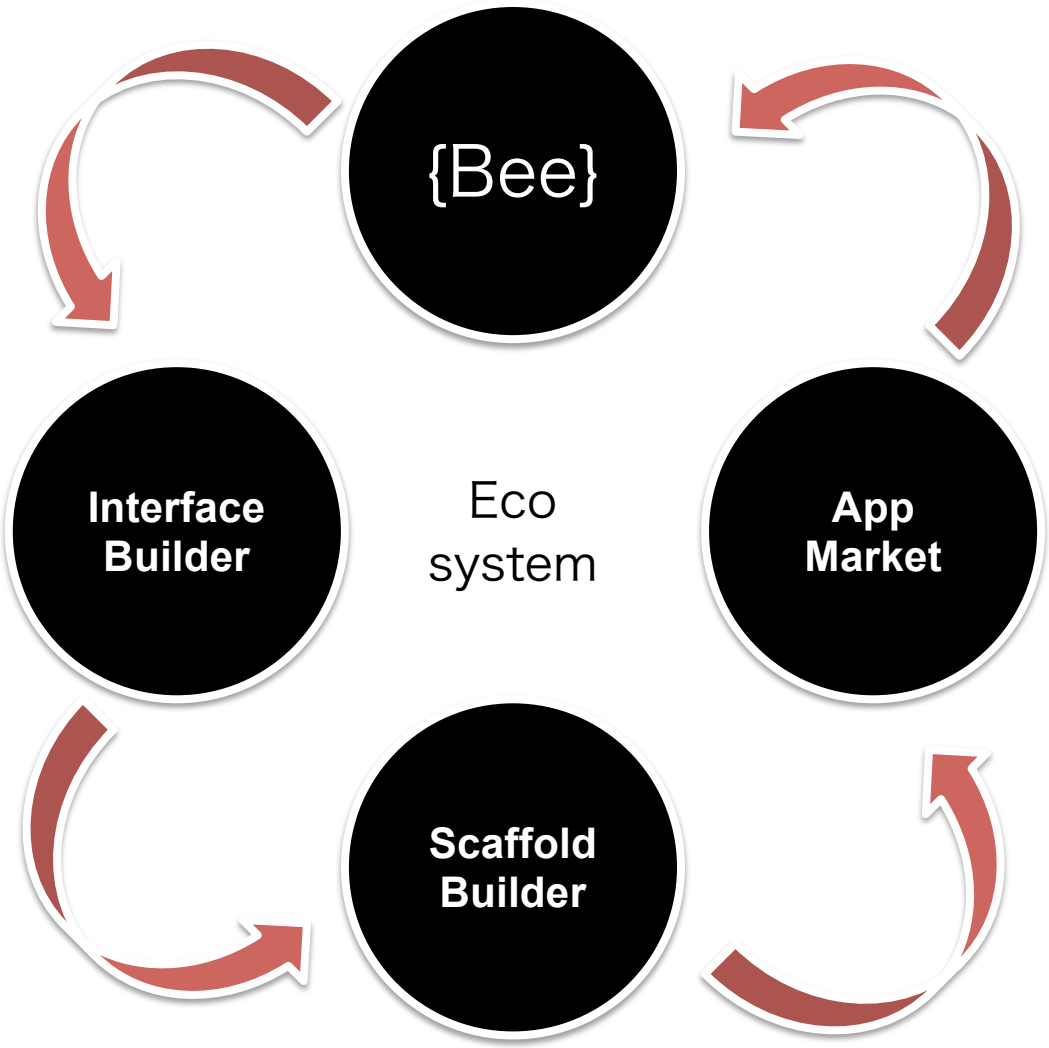


附加工具 - Scaffold

该工具用来根据 JSON schema 生成 Model 及 Controller 代码，具体说明请详见 /tools/scaffold/scaffold.md。

{Bee} 的未来

你一定会认为我们很疯狂，下着一盘很大的棋，但 {Bee} 坚持要走的路必定会是未来的趋势，必定成为 iOS 移动开发技术的领导者。围绕于技术上我们所想的一切，这里会出现更多让你意想不到的东西，以至于改变你的所想一切，在未来的数月里它即将上线。



加入我们，QQ 群号：79054681