北京信息科技大学

**毕 业 设 计（论 文）附录**

**题 目：** 基于iOS的宠物社交互动社区

系统的设计与实现

**学 院：** 计算机学院

**专 业：** 软件工程

**学生姓名：**翁培钧 **班级/学号**软工1501/2015011206

**指导老师/督导老师：** 郝保水

**起止时间：** 2019年2月25日至2019年6月21日

**目 录**

附录1：开题报告 ………………………………………………………………………………… 共7页

附录2：程序代码 ………………………………………………………………………………… 共19页

附录3：英文文献译文 …………………………………………………………………………… 共8页

附录4：英文文献原文 …………………………………………………………………………… 共10页

**附件1**

基于iOS的宠物社交互动社区系统的设计与实现

**开题报告**

**班级（学号） 软工1501（2015011206） 姓名 翁培钧**

**指导教师 郝保水**

**一、综述**

现代中国男性和中国女性搭讪时常见的一个话题：“你家里养狗/猫吗？”，从侧面反映了饲养宠物已经成为了现代中国一个较为普遍的社会现象，甚至在某些地方出现了一个家庭饲养了多只宠物的情况。宠物的饲养与社会整个人群的生活方式有较大关系，单身青年和空巢老人的日益增多让平常的宠物饲养带上了某种情感属性。数据显示，一个国家的单身青年和空巢老人越多，宠物的养殖就越热情，我国20～29岁的单身青年约有2000万人，而且我国老年化趋势也在逐步加强，正大步迈向老年化社会，这两方面原因都将促成我国市场进入到一个全新的增长高峰期。

中国宠物经济每年有将近 30% 的增长，且目前呈现出了高速增长的趋势，尤其是国内的宠物饲养从以往传统的“看家护院式”的需求到现如今“精神陪伴式”需求的重要转变之后，国人在宠物方面的开支增长将会持续增长。据中农大动物医学研究院发布的研究数据表明，当国家人均GDP达到3000-8000美元时，当国的宠物经济将会迎来迅速发展。全球宠物市场中，北美地区宠物经济总值以占据整个市场的 37% 位列第一，第二为占据24%的欧洲，而亚洲市场以23%位列第三，从近几年数据的显示来看，亚洲宠物市场已是全球宠物经济市场有力的推进器，在未来的几年时间中亚洲市场将有机会赶超欧洲从而成为全球第二大的宠物市场。

面对国内如此大的宠物市场，存在以下几个潜在问题，首先是服务，目前对于用户姑管理和宠物相关的预约、从事社交活动等需求局限在大头，很难把消息进行共享；其次是“兽医”功能，宠物主人之间分享宠物饲养的经验渠道不足，没有较好的分享渠道进行经验共享，相关知识沉淀受限；最后是电子商业领域，可以把宠物和宠物主人之间产生交互过程积累出的相关产品进行产出与零售，针对不同的宠物品种和宠物主人的差异性需求进行个性化定制，拓展与宠物相关行业的经济增长。针对与比较突出的社交活动问题，可以拓展到当地企业和市场，在这些地方的宠物主人可以搜索产品和服务，包括兽医、训练者、宠物美容师、宠物屋已经其它相关产品。

总部位于美国的FitBark开发了“智能项圈”，为宠物狗健康管理的一大利器，可以明确的看到狗狗较为细致的生理信息数据。本系统将针对国内宠物市场上的主要问题，结合目前较为成熟的技术组件，在客户端上，基于 iOS 使用 MVC 架构结合多种设计模式构建面向用户的客户端应用，基于 Django框架，搭配 Redis、Nginx和Mysql等相关组件搭建后端服务，解析出相关业务的纠结点，最终构建一套完整的具备宠物社交、健康管理等功能的产品，提供清晰便捷的行业解决方案。

**二、研究内容**

1. 研究方向：

基于iOS平台构建一套完整的具备宠物社交，提供宠物健康管理的移动应用。

iOS是目前第二大享誉全球的移动操作系统，其以安全、流畅和拥有大量优秀的应用而著称，Apple长期致力于打造操作便捷、简洁明了的操作系统，提供完善的基础平台服务，为解决各类生活中实际问题铺垫了大量技术储备，同时iOS中也暴露出了大量基于硬件加速的API，能够在解决某些耗时问题中助力[7]。

宠物的角色在如今社会正在慢慢转变，在这个转变的过程中需要有提供较为完整的基础管理服务的平台或者社区进行指导，让宠物和人能够更好的相处，让我们能够了解到宠物的实际生理情况，提供一个友好的宠物饲养社会环境。

1. 研究内容：

抽象宠物社交和健康管理等问题，利用iOS提供的多样服务完成宠物社交互动社区系统。

抽象实际生活中的问题需要对该问题有较为深刻的理解，映射到实际产品中需要较为确切的体现出原问题的主要性。宠物的社交从侧面上看实际上是宠物主人的社交，通过构建宠物主人的社交关系网来解决宠物配种等实际需求问题。

作为宠物的主人，我们需要对宠物的生理健康有较为细致的了解，今日卡路里消耗是否达到了该物种的最低要求，进食进水量是否达到标准线以上等生理指标需要宠物主人经常性的关注，而实际上这是大部分宠物主人容易忽略的问题，这会导致宠物滋生各种疾病，缩短其存活时间，进而影响到宠物主人的心理活动。

1. 系统功能：
2. 数据导入

从未饲养过宠物的用户，产品提供了领养“虚拟宠物”功能，我们对“虚拟狗”增加了真实宠物所具备的所有基本属性，能够较好的模拟出宠物在真实的日常生活中所反映出的问题，用户通过领养一只“虚拟狗”来养成对宠物饲养的基本知识和习惯。

对于已经饲养过宠物的用户，产品提供了“数据导入”的功能，可以选择真实宠物的性别、品种、生日、体重、绝育情况、感情状态金额进食进水量等基本体征数据。

1. 信息流

在首页信息流中将展示出以宠物的日常生活见闻，用户可发布与宠物有关纯文本或图文消息，展示宠物的日常生活习性，为后续寻找玩伴、配偶等社交需求提供一个较为完整的展示平台。

1. 好友系统

在产品中宠物和人同样是“一等公民”，所有的操作都围绕宠物进行，在好友系统中提升了宠物的级别，可以直接搜索宠物或宠物主人进行关注，也可以与该宠物或宠物主人发起聊天，若是对宠物建立了聊天，则消息将转发给该宠物的管理员，管理员为宠物的最高权限所有者。

1. 玩乐

提供“遛狗”和“撸猫”两个动作入口，在“遛狗”模块中，用户需携带宠物和装有产品的设备并打开GPS，一同进行遛狗行为，产品将会根据获取到的GPS数据绘制出真实的遛狗轨迹，并在遛狗结束后根据时间、距离、天气等综合因素并结合该宠物品种的标准运动量对该次遛狗评分。

在“撸猫”模块中，用户需在真实撸猫过程中打开产品并使其置于前台活跃状态，并保持尽可能长的时间，如果中途有电话等系统级应用切入前台或用户手动进入桌面而导致产品进入后台，则此次撸猫行为结束，产品会根据一次撸猫行为过程中收集到的实际数据打分。

1. 喂食

提供“进食”和“喂水”两个模块，产品均根据宠物品种的标准进食进水量结合宠

物当前录入数据计算出此时应当进食进水量为多少，给用户一个较为准确的参考值，帮助用户更加了解自己的宠物。

1. 评分

评分系统包含了宠物基本生理特征数据变化的所有过程，比如运动、进食进水、玩乐等，产品将通过评分系统展现出宠物主人在宠物饲养过程中的不足或可取之处，宠物主人可根据产品计算出的评分改进或分享自己的宠物饲养经验。

**三、实现方法及预期目标**

初步方案：

1. 学习宠物行业的基本知识[1]。包括熟悉流行宠物（如猫狗）等基本生活习性，了解其生理特征，熟悉其饲养习惯，并实地考察宠物饲养过程中的注意点和难点，以及宠物对生活环境（如天气、绿化环境、人口密度等）的适应程度，与相关从业人员和宠物主人交谈，进一步确认产品需求和最终产品方向。
2. 构建产品基本原型。包括app原型和业务原型。其中app原型包括了移动应用原型和后端架构原型，使用Sketch进行移动应用原型和高保真设计图设计。后端架构原型将以遵循MVP原则进行最小化可实行产品进行设计。业务原型将结合宠物相关产业链进行业务上下游原型搭建，初步验证产品业务的可行性。
3. 开发。开发阶段工作分为客户端开发和服务端开发，客户端基于iOS平台使用Swift和Objective-C语言及衍生框架进行开发[2]，采用MVC设计模式。服务端基于Django框架，借助Redis、Mysql、Nginx等基本技术组件进行搭建高可用的服务端[3-4]。使用遵循RESTful规范的JSON 格式[5]进行API[6]的数据传输。
4. 测试。在测试阶段主要分为alpha和beta测试[7]。在alpha测试阶段主要任务将围绕提升产品的功能、局域化、可用性、可靠性、性能和支持等指标进行。beta测试阶段产品将发布到TestFlight众测平台，该平台具备最高10000真实用户的同时测试，在该阶段中主要依据真实用户所反馈的问题进行产品的优化。
5. 发布。

重点：

1. 熟悉宠物行业基本知识

通过对宠物行业的快速学习，能够覆盖大部分行业内主要分支，从宠物繁育、检验检疫、食品安全、配种和基本社交需求等方面快速了解，能够在抽象实际问题阶段提供强有力的支持。

1. 精简且准确的操作

本人已具备熟练的iOS移动应用开发技能，但对于开发出适合覆盖年龄段跨度较大的人群，同时不但要保证产品的业务流程准确无误还要操作逻辑简洁明了的产品没有较大把握，需要反复斟酌和思考。

1. 健硕的后端服务
   1. 一个强有力的后端服务是一个产品成功的基本要素。使用已经迭代多次、反映良好的Django框架作为后端服务的主要框架是经过了调研了flask、tornado和quixote而作出的选择，Django采用了“配置大于约定”的原则，且严格遵守了MVC设计模式，方便搭建出高可用的后端服务。
   2. 采用redis作为用户鉴权token[8]的缓存服务，充分利用了redis基于内存缓存的优势，同时也迎合了几乎所有接口都需要进行token校验[9]的频繁操作，降低了
   3. 服务器压力。
   4. 采用nginx作为web服务器[10]，方便后续业务的升级迭代做负载均衡、文件缓存等功能，且便于调节服务器资源，提供稳定的服务。

难点：

1. 数据缓存
   1. 产品的属性定义了在日常的使用过程中会涉及到大量的数据交互，需要客户端提供一定的单机能力，在服务不可用等极端情况下需要保证良好的产品交互逻辑。
   2. 在部分模块中会设计到大量的交互，甚至是重复性数据交互，需要服务端提供一定的“热数据”缓存能力，能够把请求频繁的数据进行缓存处理，降低服务端运行压力。
2. 体征数据表现

对宠物体征数据进行归类计算并形成评分是产品的一大特色。关于如何良好的保证用户隐私的情况下通过特定指标数据结合自研公式计算出最终评分是决定用户留存率的一大关键问题。

1. 宠物模拟

产品涉及到众多模拟宠物真实生活习性的地方，如食物和水的消耗、遛狗撸猫等行为，需要结合宠物各个品种的标准值进行同等替换计算。

1. 信息流

信息流是构建产品社交属性的基本内容，而提供一个良好的信息流的展示是决定社交属性体现得好与坏的核心指标，产品将结合“推”“拉”两种信息流展示方式，持续优化信息流的展示。

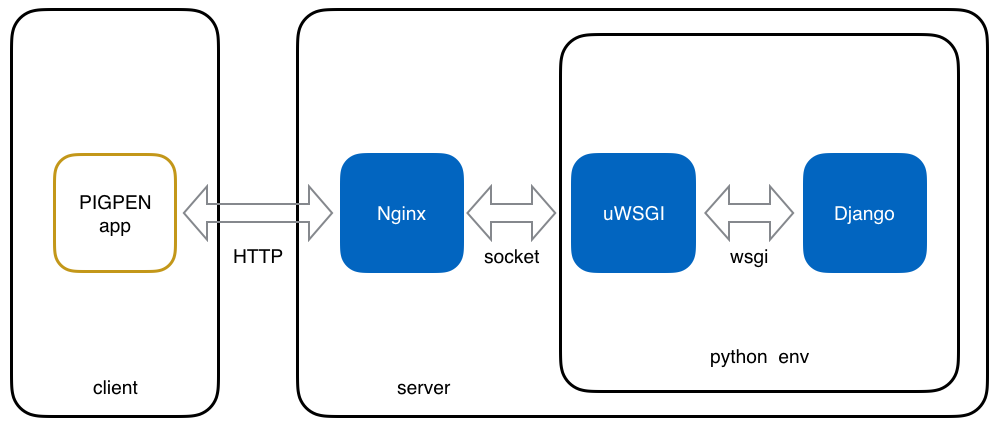


图 1 系统工作原理图

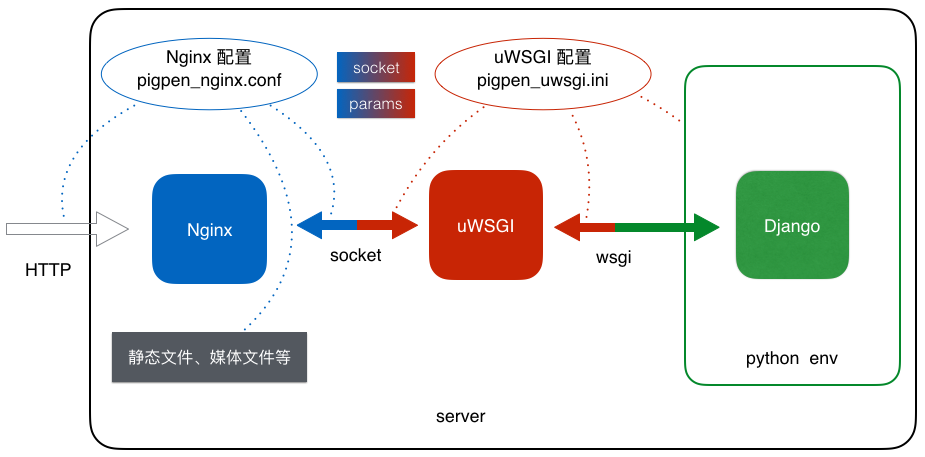


图 2 系统运行过程图

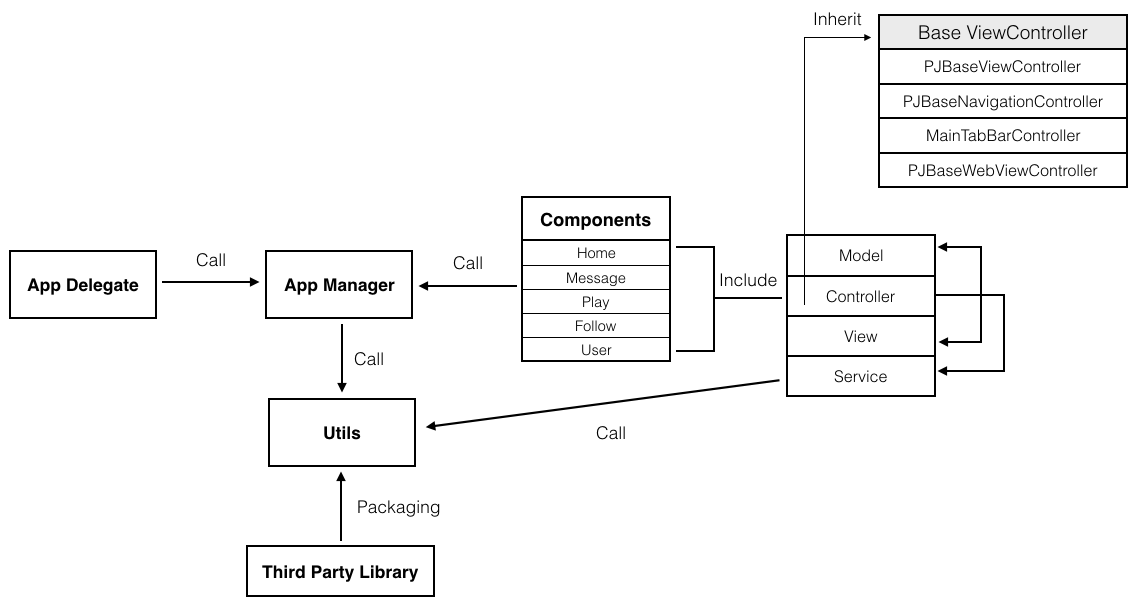


图 3 App架构图

环境：

1. 客户端

iOS 11.0及以上；

支持当前售卖的所有iPhone型号；

1. 服务端

Django 2.0.7

Ubuntu. 16.04

Redis server 4.0.11

Mysql 5.6.0

Nginx。 1.4.7

1. 开发工具

Xcode 10.1

PyCharm 2018.1.2

Postman 6.7.3

**四、对进度的具体安排**

第1-2周 ：撰写开题报告初稿，学习宠物行业基本知识；

第3 周 ：完成开题报告，进入构建产品基本原型阶段；

第4-5周 ：分析需求，进入最小化可实行产品迭代阶段；

第6周 ： 完成一期后端服务编开发，进入客户端开发；

第7-8周 ：完成一期客户端开发，进入二期服务端开发；

第9-10周 : 完成二期客户端开发，进入alpha和beta测试阶段；

第11周 : 发布；

第12—13周：进入编写毕业设计论文等相关工作阶段；

第14周：论文修改；

第15周：论文评审；

第16-17周 ：毕设答辩。

**五、参考文献**

[1] 张茂杨, 彭小凡, 胡朝兵, 等. 宠物与人类的关系: 心理学视角的探讨[J]. 心理科学进展, 2015, 23(1): 142-149.

[2] Rebouças M, Pinto G, Ebert F, et al. An empirical study on the usage of the swift programming language[C]//2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER). IEEE, 2016, 1: 634-638.

[3] Pautasso C, Zimmermann O, Leymann F. Restful web services vs. big'web services: making the right architectural decision[C]//Proceedings of the 17th international conference on World Wide Web. ACM, 2008: 805-814.

[4] Calder B, Krintz C, John S, et al. Cache-conscious data placement[C]//ACM SIGPLAN Notices. ACM, 1998, 33(11): 139-149.

[5] Bray T. The javascript object notation (json) data interchange format[R]. 2017.

[6] Bloch J. How to design a good API and why it matters[C]//Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications. ACM, 2006: 506-507.

[7] Bertolino A. Software testing research: Achievements, challenges, dreams[C]//2007 Future of Software Engineering. IEEE Computer Society, 2007: 85-103. [8] Jones M, Bradley J, Sakimura N. Json web token (jwt)[R]. 2015.

[9] 时子庆, 刘金兰, 谭晓华. 基于 OAuth2. 0 的认证授权技术[J]. 计算机系统应用, 2012, 21(3): 260-264.

[10] Chi X, Liu B, Niu Q, et al. Web load balance and cache optimization design-based Nginx under high-concurrency environment[C]//2012 Third International Conference on Digital Manufacturing & Automation. IEEE, 2012: 1029-1032.

**指导教师：**（签署意见并签字） 年 月 日

**督导教师：**（签署意见并签字） 年 月 日

**领导小组审查意见：**

审查人签字： 年 月 日

**附件2**

**程序代码**

# **用户操作相关核心代码如下所示：**

import hashlib

from django.conf import settings

from .models import MasUser

from common import token\_utils, utils, decorator, masLogger

from pet.models import Pet

from pet.views import get\_pet

@decorator.request\_methon('POST')

@decorator.request\_check\_args(['phoneNumber', 'password', 'avatar', 'gender'])

def create\_masuser(request):

phone\_number = request.POST.get('phoneNumber')

# password is a hash\_str

password = request.POST.get('password')

nick\_name = request.POST.get('nick\_name')

avatar = request.POST.get('avatar')

gender = request.POST.get('gender')

if MasUser.objects.filter(phone\_number=phone\_number).exists():

return utils.ErrorResponse(2333, 'user exist', request)

masuser = MasUser.create(phone\_number=phone\_number, password=password,

avatar=avatar, nick\_name=nick\_name, gender=gender)

token = token\_utils.create\_token(masuser.uid)

json = {

'masuser': masuser.toJSON(),

'token': token,

# 刚注册用户肯定都没有宠物和虚拟宠物

'feeding\_status': [0, 0, 0]

}

return utils.SuccessResponse(json, request)

@decorator.request\_methon('POST')

@decorator.request\_check\_args(['sign', 'timestamp', 'phoneNumber'])

def login(request):

phone\_number = request.POST.get('phoneNumber')

sign = request.POST.get('sign')

timestamp = request.POST.get('timestamp')

masuser = MasUser.objects.filter(phone\_number=phone\_number).first()

if masuser:

md5 = hashlib.md5()

md5.update((masuser.password + timestamp).encode('utf-8'))

masuser\_password\_hash = md5.hexdigest()

if sign == masuser\_password\_hash:

token = token\_utils.create\_token(masuser.uid)

json = {

'masuser': masuser.toJSON(),

'token': token,

}

return utils.SuccessResponse(json, request)

else:

return utils.ErrorResponse(2333, '密码错误', request)

else:

return utils.ErrorResponse(2333, '用户不存在', request)

@decorator.request\_methon('POST')

@decorator.request\_check\_args([])

def logout(request):

username = request.POST.get('uid')

token\_utils.delete\_token(username)

json = {

'isLogOut': 'true'

}

return utils.SuccessResponse(json, request)

# 获取用户宠物信息

@decorator.request\_methon('GET')

@decorator.request\_check\_args([])

def get\_user\_pet\_info(request):

uid = request.GET.get('uid')

# 获取真实宠物信息

real\_pet\_array = []

real\_pets = Pet.objects.filter(user\_\_uid=uid)

for real\_pet in real\_pets:

real\_pet\_array.append(get\_pet(real\_pet.pet\_id, uid))

json = {

'pets': real\_pet\_array,

}

return utils.SuccessResponse(json, request)

# 获取用户简单信息

@decorator.request\_methon('GET')

@decorator.request\_check\_args(['details\_uid'])

def get\_user\_details(request):

details\_uid = request.GET.get('details\_uid')

user = MasUser.objects.filter(uid=details\_uid).first()

if user:

json = {

'masuser': user.toJSON(),

}

return utils.SuccessResponse(json, request)

else:

return utils.ErrorResponse(2333, 'user not exist', request)

@decorator.request\_methon('POST')

@decorator.request\_check\_args(['avatar', 'gender'])

def update\_user(request):

uid = request.POST.get('uid')

nick\_name = request.POST.get('nick\_name')

avatar = request.POST.get('avatar')

user = MasUser.objects.filter(uid=uid).\

update(avatar=avatar, nick\_name=nick\_name).first()

if user:

json = {

'masuser': user.toJSON()

}

return utils.SuccessResponse(json, request)

else:

return utils.ErrorResponse(2333, '用户不存在', request)

@decorator.request\_methon('GET')

@decorator.request\_check\_args([])

def update\_token(request):

uid = request.GET.get('uid')

token\_utils.delete\_token(uid)

token = token\_utils.create\_token(uid)

json = {

'token': token,

}

return utils.SuccessResponse(json, request)

@decorator.request\_methon('GET')

@decorator.request\_check\_args(['phoneNumber'])

def check\_phone(request):

phone\_number = request.GET.get('phoneNumber')

user = MasUser.objects.filter(phone\_number=phone\_number).first()

if user:

masLogger.log(request, 2333, '用户已注册')

return utils.ErrorResponse(2333, '用户已注册', request)

else:

json = {

'status': '可注册'

}

return utils.SuccessResponse(json, request)

@decorator.request\_methon('GET')

@decorator.request\_check\_args([])

def getRCToken(request):

from rongcloud import RongCloud

uid = request.GET.get('uid')

nick\_name = request.GET.get('nick\_name')

app\_key = settings.RC\_APP\_KEY

app\_secret = settings.RC\_APP\_SECRET

rcloud = RongCloud(app\_key, app\_secret)

r = rcloud.User.getToken(userId=uid,

name=nick\_name,

portraitUri='https://avatars0.githubusercontent.com/u/15074681?s=460&v=4')

r\_json = eval(str(r.response.content, encoding='utf-8'))

if r\_json['code'] == 200:

json = {

'token': r\_json['token']

}

return utils.SuccessResponse(json, request)

else:

masLogger.log(request, 2333, str(r.response.content, encoding='utf-8'))

return utils.ErrorResponse(2333, 'RCToken error', request)

**宠物相关操作核心代码如下所示：**

import pinyin

from django.conf import settings

from common import decorator, masLogger

from user.models import MasUser

from common import utils

from .models import Pet, dog\_breed, cat\_breed

from relationship.models import PetRelationship

from avatar.models import Avatar

@decorator.request\_methon('POST')

@decorator.request\_check\_args(['pet\_nick\_name', 'gender', 'pet\_type',

'birth\_time', 'weight', 'ppp\_status',

'love\_status', 'relation\_code', 'avatar\_key',

'breed\_type', 'food\_weight', 'activity'])

def create\_pet(request):

pet\_nick\_name = request.POST.get('pet\_nick\_name', '')

gender = request.POST.get('gender', '')

pet\_type = request.POST.get('pet\_type', '')

birth\_time = request.POST.get('birth\_time', '')

weight = request.POST.get('weight', '')

ppp\_status = request.POST.get('ppp\_status', '')

love\_status = request.POST.get('love\_status', '')

relation\_code = request.POST.get('relation\_code', -1)

uid = request.POST.get('uid', '')

avatar\_key = request.POST.get('avatar\_key')

breed\_type = request.POST.get('breed\_type')

food\_weight = request.POST.get('food\_weight')

activity = request.POST.get('activity')

user = MasUser.objects.filter(uid=uid).first()

if user:

# 宠物实体

pet = Pet.create(pet\_nick\_name=pet\_nick\_name,

gender=gender,

pet\_type=pet\_type,

weight=weight,

birth\_time=birth\_time,

love\_status=love\_status,

ppp\_status=ppp\_status,

user=user,

breed\_type=breed\_type,

food\_weight=food\_weight,

activity=activity)

# 宠物关系实体

relation = PetRelationship(pet\_id=pet.pet\_id, uid=uid,

relationship\_code=relation\_code)

relation.save()

# 宠物头像

Avatar(own\_id=pet.pet\_id, avatar\_key=avatar\_key).save()

petJSON = pet.toJSON()

petJSON['avatar\_url'] = utils.create\_full\_image\_url([avatar\_key])[0]

petJSON['relationship'] = int(relation.relationship\_code)

return utils.SuccessResponse(petJSON, request)

else:

return utils.ErrorResponse('2333', 'user not exist', request)

@decorator.request\_methon('GET')

@decorator.request\_check\_args(['pet\_type'])

def get\_breeds(request):

pet\_type = request.GET.get('pet\_type', '')

functions = {

'0': cat(),

'1': dog()

}

if pet\_type in functions.keys():

json = {

'breeds': functions[pet\_type]

}

return utils.SuccessResponse(json, request)

else:

return utils.ErrorResponse('2333', '不支持该物种', request)

@decorator.request\_methon('GET')

@decorator.request\_check\_args(['imageCount'])

def get\_pet\_upload\_avatar\_token(request):

imageCount = int(request.GET.get('imageCount', "1"))

key = 'pet\_avatar'

jsons = utils.create\_upload\_image\_token(imageCount, key)

f\_json = {

# list 倒置：不写区间范围的话，默认为原list,因此L[:]和L[::]都表示原list。

# 根据以上推算，想要倒置list,只需要对原list取负步距-1，即每次回退一个即可得到

# from: https://blog.csdn.net/akisayaka/article/details/50042175

'upload\_tokens': jsons[::-1]

}

return utils.SuccessResponse(f\_json, request)

@decorator.request\_methon('POST')

@decorator.request\_check\_args(['keys'])

def upload\_pet\_avatar\_key(request):

keys = request.POST.get('keys')

keys\_array = keys.split(',')

if keys:

urls = utils.create\_full\_image\_url(keys\_array)

json = {

'image\_urls': urls

}

return utils.SuccessResponse(json, request)

else:

return utils.ErrorResponse('2333', 'keys is empty', request)

def get\_pet(pet\_id, uid):

"""

获取某一宠物全部信息

:param pet\_id: 宠物 id

:param uid: 用户 id

:return:

"""

pet = Pet.objects.filter(pet\_id=pet\_id).first()

if pet:

pet\_avatar\_key = Avatar.objects.filter(own\_id=pet\_id).first()

if pet\_avatar\_key:

key = pet\_avatar\_key.avatar\_key

pet\_avatar\_url = utils.create\_full\_image\_url([key])[0]

pet\_relation = PetRelationship.objects.filter(pet\_id=pet\_id, uid=uid)\

.first()

if pet\_relation:

petJSON = pet.toJSON()

petJSON['avatar\_url'] = pet\_avatar\_url

petJSON['relationship'] = pet\_relation.relationship\_code

return petJSON

# 获取所有狗品种

def dog():

dog\_breeds = dog\_breed.objects.all()

# 所有种类

breeds = []

# 当前种类名

breed\_groups = []

group = "A"

for breed in dog\_breeds:

if breed.group != group:

breed\_group = {

'group': group,

'breeds': breed\_groups,

}

breeds.append(breed\_group)

group = breed.group

breed\_groups = []

b\_group = {

'id': breed.pk,

'zh\_name': breed.zh\_name,

}

breed\_groups.append(b\_group)

return breeds

# 获取所有猫品种

def cat():

cat\_breeds = cat\_breed.objects.all()

# 所有种类

breeds = []

# 当前种类名

breed\_groups = []

group = "A"

for breed in cat\_breeds:

if breed.group != group:

breed\_group = {

'group': group,

'breeds': breed\_groups,

}

breeds.append(breed\_group)

group = breed.group

breed\_groups = []

b\_group = {

'id': breed.pk,

'zh\_name': breed.zh\_name,

}

breed\_groups.append(b\_group)

return breeds

# 初始化：尽量通过 python shell 调用该方法

def init\_dog\_breed():

f = open(settings.DOG\_BREED\_DIR, 'r')

f\_str = f.read()

f\_str\_arr = f\_str.split()

# 删除 array 中的第一个 'A'

del f\_str\_arr[0]

group = 'A'

for dog\_name in f\_str\_arr:

first\_cat\_name = pinyin.get(dog\_name, format='strip')[0:1].upper()

if first\_cat\_name != group:

group = first\_cat\_name

# 切换 group 时跳过

continue

dog\_breed(zh\_name=dog\_name, group=group).save()

f.close()

# 新增狗品种

def add\_dog\_breed(breed\_name='未知品种', group='W'):

dog\_breed(zh\_name=breed\_name, group=group).save()

# 初始化：尽量通过 python shell 调用该方法

def init\_cat\_breed():

f = open(settings.CAT\_BREED\_DIR, 'r')

f\_str = f.read()

f\_str\_arr = f\_str.split()

# 删除 array 中的第一个 'A'

del f\_str\_arr[0]

group = 'A'

for cat\_name in f\_str\_arr:

first\_cat\_name = pinyin.get(cat\_name, format='strip')[0:1].upper()

if first\_cat\_name != group:

group = first\_cat\_name

# 切换 group 时跳过

continue

cat\_breed(zh\_name=cat\_name, group=group).save()

f.close()

# 新增猫品种

def add\_cat\_breed(breed\_name='未知品种', group='W'):

cat\_breed(zh\_name=breed\_name, group=group).save()

**推文相关操作核心代码如下所示：**

from django.contrib.contenttypes.models import ContentType

from .models import Blog

from user.models import MasUser

from read\_statistics.models import ReadNumber

from like\_statistics.models import LikeCount

from comment.models import Comment

from common import utils, decorator, masLogger

@decorator.request\_methon('POST')

@decorator.request\_check\_args(['content'])

def create\_blog(request):

masuserId = request.POST.get('masuser\_id', '')

content = request.POST.get('content', '')

masuser = MasUser.objects.filter(pk=masuserId).first()

if masuser:

return utils.ErrorResponse(2333, '用户不存在', request)

Blog(content=content, masuser=masuser).save()

return utils.SuccessResponse('发布成功', request)

@decorator.request\_methon('GET')

@decorator.request\_check\_args(['page'])

def blog\_list(request):

page\_num = request.GET.get('page')

blogs = utils.get\_page\_blog\_list(Blog.objects.filter(is\_deleted=0).

values(), page\_num)

final\_blogs = []

for blog in blogs:

masuserId = blog['masuser\_id']

masuser = MasUser.objects.get(pk=masuserId)

# replace field `masuser`

blog['masuser'] = masuser.toJSON()

# get blog read\_num

content\_type = ContentType.objects.get(model='blog')

readnum, create = ReadNumber.objects.get\_or\_create(

content\_type=content\_type, object\_id=blog['id'])

blog['read\_num'] = readnum.read\_num

final\_blogs.append(blog)

json = {

'blogs': list(final\_blogs),

}

return utils.SuccessResponse(json, request)

@decorator.request\_methon('GET')

@decorator.request\_check\_args(['blog\_id'])

def delete\_blog(request):

masuser\_id = request.GET.get('masuser\_id')

blog\_id = request.GET.get('blog\_id')

blog = Blog.objects.get(pk=blog\_id)

# 记得 string to int

if blog.masuser.pk == int(masuser\_id):

findBlog = Blog.objects.filter(pk=blog\_id)

if findBlog.is\_deleted == 0:

findBlog.delete()

return utils.SuccessResponse('删除成功', request)

else:

return utils.ErrorResponse(2333, '删除失败，文章不存在', request)

else:

return utils.ErrorResponse(2333, '删除失败，只能删除自己发布的文章', request)

@decorator.request\_methon('GET')

@decorator.request\_check\_args(['page'])

def get\_user\_blog(request):

userId = request.GET.get('masuser\_id')

page\_num = request.GET.get('page')

blogs = utils.get\_page\_blog\_list(Blog.objects.filter(

masuser\_\_pk=userId, is\_deleted=0), page\_num)

final\_blogs = []

for blog in blogs:

b = {

'id': blog.pk,

'content': blog.content,

'created\_time': blog.created\_time,

}

final\_blogs.append(b)

if blogs:

json = {

'blogs': list(final\_blogs)

}

return utils.SuccessResponse(json, request)

else:

return utils.ErrorResponse(2333, '该用户未发布文章', request)

@decorator.request\_methon('GET')

@decorator.request\_check\_args(['content\_type', 'object\_id'])

def blog\_details(request):

content\_type = request.GET.get('content\_type', '')

# blog\_id

object\_id = request.GET.get('object\_id', '')

contentType = ContentType.objects.get(model=content\_type)

readnum, create = ReadNumber.objects.get\_or\_create(

content\_type=contentType, object\_id=object\_id)

readnum.read\_num += 1

readnum.save()

blog = Blog.objects.filter(pk=object\_id, is\_deleted=0).filter()

if blog:

# get blog like\_num

like\_count, created = LikeCount.objects.get\_or\_create(

content\_type=contentType, object\_id=object\_id)

# get comment\_num

comments = Comment.objects.filter(content\_type=contentType,

object\_id=object\_id).count()

json = {

'blog': {

'read\_num': readnum.read\_num,

'comment\_num': comments,

'like\_num': like\_count.liked\_num,

'blog\_content': blog.content,

'blog\_created\_time': blog.created\_time.timestamp(),

},

'masuser': blog.masuser.toJSON(),

}

return utils.SuccessResponse(json, request)

else:

return utils.ErrorResponse(2333, '该文章不存在', request)

**评论相关操作核心代码如下所示：**

from django.contrib.contenttypes.models import ContentType

from user.models import MasUser

from .models import Comment

from common import utils, decorator, masLogger

@decorator.request\_methon('POST')

@decorator.request\_check\_args(['content\_type', 'content\_id', 'text'])

def create\_comment(request):

masuserId = request.POST.get('masuser\_id', '')

text = request.POST.get('text', '')

content\_type = request.POST.get('content\_type', '')

content\_id = request.POST.get('content\_id', '')

masuser = MasUser.objects.get(pk=masuserId)

model\_class = ContentType.objects.get(model=content\_type).model\_class()

model\_obj = model\_class.objects.get(pk=content\_id)

comment = Comment(text=text, masuser=masuser, content\_object=model\_obj)

parent\_id = request.POST.get('parent\_id', '')

if parent\_id:

parent = Comment.objects.get(pk=parent\_id)

if parent:

comment.parent = parent

# 设置当前评论的顶级评论

if parent.root:

comment.root = parent.root

else:

comment.root = parent

comment.save()

return utils.SuccessResponse('评论发布成功', request)

@decorator.request\_methon('POST')

@decorator.request\_check\_args(['content\_type', 'content\_id', 'page'])

def get\_comment(request):

contentType = request.POST.get('content\_type')

content\_id = request.POST.get('content\_id')

page\_num = request.POST.get('page')

content\_type = ContentType.objects.get(model=contentType)

comments = Comment.objects.filter(content\_type=content\_type,

object\_id=content\_id, parent=None)

parent\_comments = utils.get\_page\_blog\_list(comments, page\_num)

final\_comments = []

for comment in parent\_comments:

# get child comment

# MARK: do paginator

child\_comments = Comment.objects.filter(content\_type=content\_type,

object\_id=content\_id,

root=comment)\

.order\_by('comment\_time')

child\_final\_comments = []

if child\_comments:

for c\_m in child\_comments:

reply\_to = c\_m.parent.masuser.nick\_name

cc = {

'comment\_id': c\_m.pk,

'comment\_content': c\_m.text,

'comment\_created\_time': c\_m.comment\_time.timestamp(),

'masuser': MasUser.objects.get(pk=c\_m.masuser.pk).toJSON(),

'reply\_to\_masuser': {

'nick\_name': reply\_to,

}

}

child\_final\_comments.append(cc)

c\_masuser = MasUser.objects.get(pk=comment.masuser.pk)

c = {

'comment\_id': comment.pk,

'comment\_content': comment.text,

'comment\_created\_time': comment.comment\_time.timestamp(),

'masuser': c\_masuser.toJSON(),

'child\_comments': list(child\_final\_comments)

}

final\_comments.append(c)

json = {

'comments': list(final\_comments)

}

return utils.SuccessResponse(json, request)

@decorator.request\_methon('POST')

@decorator.request\_check\_args(['content\_type', 'content\_id'])

def delete\_comment(request):

masuser\_id = request.POST.get('masuser\_id', '')

contentType = request.POST.get('content\_type', '')

content\_id = request.POST.get('content\_id', '')

content\_type = ContentType.objects.get(model=contentType)

masuser = MasUser.objects.get(pk=masuser\_id)

Comment.objects.filter(content\_type=content\_type,

pk=content\_id,

masuser=masuser).delete()

return utils.SuccessResponse('删除成功', request)

**好友相关操作核心代码如下所示：**

from django.db.models import Q

from haystack.query import SearchQuerySet

from common import utils, decorator

from friend.models import Friend

from user.models import MasUser

@decorator.request\_methon('GET')

@decorator.request\_check\_args(['type'])

def getFriend(request):

uid = request.GET.get('uid')

type = request.GET.get('type')

frinds = Friend.objects.filter(Q(userA\_\_uid=uid) | Q(userB\_\_uid=uid),

status=1)

my\_frinds = []

for frind in frinds:

if frind.userA.uid == uid:

my\_frind = frind.userB

else:

my\_frind = frind.userA

if type == 0:

my\_frinds.append(my\_frind.toJSON())

else:

# 这里需要带上宠物信息

my\_frinds.append(my\_frind.toJSON())

json = {

'friends': my\_frinds,

}

return utils.SuccessResponse(json, request)

@decorator.request\_methon('POST')

@decorator.request\_check\_args(['friendId', 'status'])

def addFriend(request):

uid = request.POST.get('uid')

friendId = request.POST.get('friendId')

status = request.POST.get('status')

uid\_user = MasUser.objects.filter(uid=uid).first()

friendId\_user = MasUser.objects.filter(uid=friendId).first()

if uid\_user and friendId\_user:

if Friend.objects.filter(Q(userA\_\_uid=uid)

| Q(userB\_\_uid=uid)

| Q(userA\_\_uid=uid)

| Q(userB\_\_uid=uid),

status=1):

return utils.ErrorResponse(2333,

"You're already friends",

request)

else:

Friend(userA=uid\_user,

userB=friendId\_user,

status=status).save()

json = {

'status': 'ok'

}

return utils.SuccessResponse(json,

request)

else:

return utils.ErrorResponse(2333,

'user not exist',

request)

@decorator.request\_methon('GET')

@decorator.request\_check\_args(['s\_nick\_name'])

def searchFriend(request):

nick\_name = request.GET.get('s\_nick\_name')

users = SearchQuerySet().models(MasUser).filter(nick\_name\_\_contains=nick\_name)

f\_users = []

for user in users:

if user.object:

f\_users.append(user.object.toJSON())

json = {

'users': f\_users

}

return utils.SuccessResponse(json, request)

**点赞相关操作核心代码如下所示：**

from django.contrib.contenttypes.models import ContentType

from django.db.models import ObjectDoesNotExist

from .models import LikeCount, LikeRecord

from user.models import MasUser

from blog.models import Blog

from common import utils, decorator, masLogger

@decorator.request\_methon('POST')

@decorator.request\_check\_args(['content\_type', 'object\_id', 'is\_like'])

def like\_change(request):

masuser\_id = request.POST.get('masuser\_id', '')

is\_like = request.POST.get('is\_like', '')

content\_type = request.POST.get('content\_type')

object\_id = int(request.POST.get('object\_id'))

masuser = MasUser.objects.get(pk=masuser\_id)

# is\_blog

try:

content\_type = ContentType.objects.get(model=content\_type)

model\_class = content\_type.model\_class()

model\_obj = model\_class.objects.get(pk=object\_id)

except ObjectDoesNotExist:

return utils.ErrorResponse(2333, '点赞文章不存在')

if is\_like == 'true':

# will like

like\_record, created = LikeRecord.objects.get\_or\_create(

content\_type=content\_type, object\_id=object\_id, masuser=masuser)

if created:

# not like, to like

# like\_num + 1

like\_count, created = LikeCount.objects.get\_or\_create(

content\_type=content\_type, object\_id=object\_id)

like\_count.liked\_num += 1

like\_count.save()

json = {

'blog\_id': object\_id,

'like\_num': like\_count.liked\_num,

}

return utils.SuccessResponse(json, request)

else:

# did like, not like

return utils.ErrorResponse(2333, '已点赞过该文章', request)

else:

# cancle like

if LikeRecord.objects.filter(content\_type=content\_type,

object\_id=object\_id,

masuser=masuser).exists():

# did like, cancle like

like\_record = LikeRecord.objects.get(content\_type=content\_type,

object\_id=object\_id,

masuser=masuser)

like\_record.delete()

# like\_num - 1

like\_count, created = LikeCount.objects.get\_or\_create(

content\_type=content\_type, object\_id=object\_id)

if not created:

like\_count.liked\_num -= 1

like\_count.save()

return utils.SuccessResponse(like\_count.liked\_num, request)

else:

return utils.ErrorResponse(2333, '数据异常', request)

else:

return utils.ErrorResponse(2333, '未点赞过该文章', request)

@decorator.request\_methon('GET')

@decorator.request\_check\_args(['page'])

def get\_like\_blog(request):

masuser\_id = request.GET.get('masuser\_id')

page\_num = request.GET.get('page')

masuser = MasUser.objects.get(pk=masuser\_id)

likes = LikeRecord.objects.filter(masuser=masuser)

like\_records = utils.get\_page\_blog\_list(likes, page\_num)

final\_likes = []

for like in like\_records:

blog = Blog.objects.get(pk=like.object\_id)

l = {

'masuser': masuser.toJSON(),

'content': blog.content,

'created\_time': blog.created\_time.timestamp()

}

final\_likes.append(l)

json = {

'blogs': list(final\_likes)

}

return utils.SuccessResponse(json, request)

**附件3**

**外文文献译文**

出处：Apple 开发者文档

**iOS中的视图编程指导**

在iOS中，我们会使用多个 window 和多个 View 在屏幕上展示你的应用程序内容。Window 本身不会包涵任何的可见内容，但提供了一个基础容器给我们应用程序中的 view 使用。View 定义了一部分你想要填入 window 中的内容。举个例子，你可以有用于显示图片、文本、形状或者以上内容的组合 view，甚至你还可以使用 view 去组织和管理其它 view。

### 概览

每一个应用程序至少拥有一个 window 和 view 去展示它的内容。UIKit 框架和其它系统框架预先定义了一些 view 供你去展示你的内容。这些 view 涵盖了简单的 button 和文本 lable 甚至是复杂的 view，比如 table view、picker view 和 scroll view。在这些预先定义好的 view 中如果没有提供你想要的 view，你可以自定义 view 并自己管理绘制和事件处理。

### view 管理你的应用程序可视内容

一个 view 是 UIView 类（或者它的一个子类）的实例并管理了一个应用程序 window 中的矩形区域。view 能够进行内容的绘制，处理多点触控事件和管理该 views 中子视图的约束。在一个矩形 view 区域中绘图会涉及到使用图像技术，比如 Core Graphics、OpenGL ES 或者 UIKit 去绘制形状、图片和文本。view能够直接响应手势识别或触摸事件。在 view 的视图层级中，父视图负责其子视图的位置和大小，且还能动态的完成。父视图的这个动态修改的能力能够让你的 view 去适应条件的改变，比如说界面旋转和动画。

你可以认为 view 是构建你的用户界面中的一个砖块。你会经常的使用一系列的 view 去构建一个等级视图去展示内容，而不是只使用一个 view。每一个在等级视图中的 view 都表现了用户界面中的一部分，并且通常是用于优化一个具体的内容类型，例如，UIKit 有很多个用于优化图像、文本和其它类型内容的 view。

相关章节：[View and Window Architecture](https://developer.apple.com/library/archive/documentation/WindowsViews/Conceptual/ViewPG_iPhoneOS/WindowsandViews/WindowsandViews.html#//apple_ref/doc/uid/TP40009503-CH2-SW1)，[Views](https://developer.apple.com/library/archive/documentation/WindowsViews/Conceptual/ViewPG_iPhoneOS/CreatingViews/CreatingViews.html#//apple_ref/doc/uid/TP40009503-CH5-SW1)

### 配合 window 显示你的 view

window 是 UIWindow 类的实例且能够处理全部呈现在应用程序用户界面上的内容。window 使用 view（及其视图控制器）来管理视图层级的交互和改变。在大部分的时候，你的应用程序 window 并不会发生改变。window 创建之后，它不会发生改变，只有通过它显示view 才会发生改变。每一个应用程序都至少拥有一个 window 在设备的主屏幕上去显示这个应用程序的用户界面。如果有外部显示设备连接到了这个设备，应用程序能够很好的在这个外部显示设备上创建第二个 window 去呈现内容。

相关章节：[Windows](https://developer.apple.com/library/archive/documentation/WindowsViews/Conceptual/ViewPG_iPhoneOS/CreatingWindows/CreatingWindows.html#//apple_ref/doc/uid/TP40009503-CH4-SW1)

### 动画提供了用户可见界面改变的反馈

动画提供了关于用户可见视图层级改变的反馈。系统定义了呈现模态视图（或单一视图）和位于不同组合视图之间的标准过渡动画。但是，view 的许多属性可以进行动画。例如，可以通过动画改变 view 的透明度，在屏幕中的位置，大小，背景颜色或者其它的属性。如果你直接使用底层的 Core Animation 层对象，你能够作出很多不错的动画。

相关章节：[Animations](https://developer.apple.com/library/archive/documentation/WindowsViews/Conceptual/ViewPG_iPhoneOS/AnimatingViews/AnimatingViews.html#//apple_ref/doc/uid/TP40009503-CH6-SW1)

### Interface Builder 扮演的角色

Interface Builder 是一个图形构造和配置你的应用程序 window 和 view 的工具。使用Interface Builder 能够把 view 集中在 nib 文件中进行处理，nib是一个存储了你可以自由操控版本的 view 或其它对象资源的文件，当你在 runtime 中加载一个 nib 文件时，你可以使用你的代码去操纵这些位于真实对象中的 view。

当你不得不去进行创建应用程序界面的工作时，使用 Interface Builder 会变得非常简单 。因为在 iOS 中已经整合进了对 Interface Builder 和 nib 的支持，只需要一点时间就可以把你应用程序的设计工作合并到 nib 中。

关于如何使用 Interface Builder 的其它信息，可查看 [Interface Builder User Guide](https://developer.apple.com/library/archive/documentation/DeveloperTools/Conceptual/IB_UserGuide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40005344)。关于如何使用 view controller 管理包涵这些 view 的 nib 文件，可查看 [View Controller Programming Guide for iOS](https://developer.apple.com/library/content/featuredarticles/ViewControllerPGforiPhoneOS/index.html#//apple_ref/doc/uid/TP40007457) 中的创建自定义视图控制器部分。

### 查看更多

因为视图是非常复杂和灵活的对象，在一个文档中完全讲解是很困难的。但是，下面的这些文档能够很好的帮助你去学习关于如何管理视图交互和用户界面的内容。

* 视图控制器是管理应用中的视图很重要的一部分。视图控制器控制着在其单一视图层级之中的所有视图，并协助这些视图在屏幕上的显示。查看 [View Controller Programming Guide for iOS](https://developer.apple.com/library/archive/featuredarticles/ViewControllerPGforiPhoneOS/index.html#//apple_ref/doc/uid/TP40007457) 来获得更多关于视图控制器的内容以及它运行规则的信息。
* 视图是应用的手势和触摸事件的关键接收者。查看 [Event Handling Guide for iOS](vscode-resource://Users/pjhubs/Documents/project/case/iOS-Course/%E9%A1%B9%E7%9B%AE/%E7%BF%BB%E8%AF%91%E2%80%94%E2%80%94ViewsprogrammingGuideforiOS.md) 来获得更多关于如何使用手势识别器和处理触摸事件。
* 自定义视图必须要使用可信赖的绘制技术去渲染其内容。查看 [Drawing and Printing Guide for iOS](https://developer.apple.com/library/archive/documentation/2DDrawing/Conceptual/DrawingPrintingiOS/Introduction/Introduction.html#//apple_ref/doc/uid/TP40010156) 来获得更多关于使用这些技术去绘制你的视图内容。
* 在一些标准视图动画无法满足的部分，你可以使用核心动画库 Core Animation。查看 [Core Animation Programming Guide](https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CoreAnimation_guide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40004514) 来获得更多关于使用核心动画库 Core Animation 实现动画的内容。

### view 和 window 的结构

view 和 window 展示应用程序用户界面和处理界面上的交互。UIKit 和其它系统框架提供了一些只需要做一点修改或完全不修改就可以使用的 view。你还可以自定义视图去展示标准视图不允许的位置内容。

### 视图结构的基础知识

我们想要通过可视化完成的事情大多都是基于 view 对象的，也就是 UIView 类的实例。view 对象在屏幕上定义了一个矩形区域，该区域能够绘制并和响应触摸事件。view 能够作为其它 view 的父容器，管理它们的位置和大小。UIView 类本身管理了这些工作的大部分，但我们也可以根据需要自定义视图之间的默认行为。

view 与 Core Animation 层共同处理内容的渲染和动画。每一个 view 都拥有一个 layer 对象（通常是 CALayer 的实例），它负责 view 的渲染内容和与 view 相关的动画。我们大多数情况下的操作应该通过 UIView 提供的接口进行，但在某些需要对渲染或动画行为进行更多的控制时，我们可以替换为对 layer 进行操作。

为了便于我们理解 view 和 layer 的关系，下面这个例子可以提供帮助。图 1-1 展示了来自于 [ViewTransitions](https://developer.apple.com/library/archive/samplecode/ViewTransitions/Introduction/Intro.html#//apple_ref/doc/uid/DTS40007411) 这个简单应用的视图结构和它与底层 Core Animation 层的关系。这个应用程序的视图包括一个 window（本质上是个 UIView），它是一个继承于 UIView 类且作为 view 容器的对象，一个图像 view，一个用于显示按钮的 toolbar 以及一个 bar button item（它不是 view，但其内部有一个可共使用的 view）。（实际上 [ViewTransitions](https://developer.apple.com/library/archive/samplecode/ViewTransitions/Introduction/Intro.html#//apple_ref/doc/uid/DTS40007411) 这个简单应用还包括了一个用于实现变换的图像 view，但为了保证简单，在图 1-1 中并没有描述出来）。每个 view 都有与之匹配的 layer 对象，可以通过 view 的 layer 属性来访问它。（因为 bar buttom item 不是 view，所以你不能直接访问它的 layer）layer 层对象的背后是 Core Animation 渲染对象，并且最终用于管理硬件缓冲区中显示在屏幕上的实际字节。

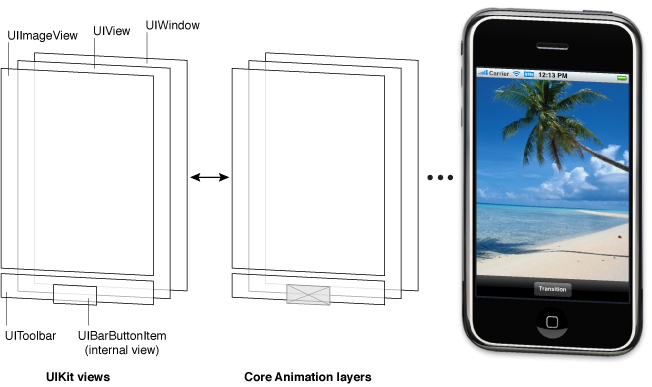


图1-1 视图层级

使用 Core Animation layer 对象对性能提升有很大的帮助。要尽可能的少调用视图对象的绘制部分代码，当这部分代码被调用时，其结果会被 Core Animation 进行缓存，并在未来进行多次重用。在需要更新视图时重用渲染出内容的会消耗大量的绘制周期。在动画中重用这部分缓存的内容是非常重要的，因为这些内容是可被操控的。使用这些缓存的内容会比创建新内容成本小很多。

### view 的视图层级以及管理子视图

除了提供自身显示的内容，view 还可以作为其它 view 的容器。当一个 view 包含了另外一个 view，父子关系就在其之中建立了。在这个父子关系中，子视图通常被称为 subView 父视图被称为 superView。这种关系类型的创建对应用程序的可视化内容和行为都造成了影响。

看上去子视图中的内容会遮挡其父视图的部分或全部内容，当子视图是非透明时，则其所占据的区域将会完全遮挡父视图。如果子视图是部分透明的，则父子视图中的内容会先进行融合再显示于屏幕上。每个 superView 都一个存储 subView 的数组，数组中元素的位置会影响每个 subview 的可见性。 当同为一个 superView 的两个 subView 互相重叠在了一起，最后添加进 superView 中的（或移动到 subView 数组最后的） subView 将会显示在在上面。

superView 与 subView 的关系还影响到了一系列的视图行为。改变一个父视图的大小时会波及到其子视图的大小和位置。当改变父视图的大小时，我们可以适当的通过一些配置方法来重新设置每个子视图大小。例如隐藏父视图，修改父视图的透明度或者对父视图的坐标系统做数学变换（3D）等这些事件同样会影响子视图。

对视图层级的排布还意味着应用程序如何响应各种事件。当一个特殊的视图发生了触摸，系统立即将这个触摸信息发送给这个视图进行处理。但是如果该视图没有对这个事件进行单独处理，则系统将会把该事件对象传递给父视图。如果父视图同样没有处理这个事件，系统将会把该事件继续传递给父视图，在整个响应链上以此类推。

更多关于如何创建视图层级的内容，可查看 [Creating and Managing a View Hierarchy](https://developer.apple.com/library/archive/documentation/WindowsViews/Conceptual/ViewPG_iPhoneOS/CreatingViews/CreatingViews.html#//apple_ref/doc/uid/TP40009503-CH5-SW47)。

### 视图绘制周期

UIView 类使用了“按需变化”的绘制模型去展示内容。当视图第一次呈现在屏幕上时，系统要求其进行内容绘制。系统对渲染内容进行快照捕获并且使用该快照作为该视图的视觉呈现。如果你从未对视图内容进行修改，执行视图内容绘制的代码将不会再执行。对视图进行的大多数操作会重复使用快照图片替代。如果你对内容进行了修改，需要通知系统视图已经修改。视图将会重复进行绘制视图内容的过程，并捕获新的图像内容为快照。

当你的视图内容修改时，你不必立即重绘这些修改的内容。相反，你可以使用 setNeedsDisplay 或 setNeedsDisplayInRect 方法使视图内容失效。这些方法将会告诉系统视图的内容已经修改，并且需要在下个时机中进行重绘。系统将会等待到当前 runloop 结束后才进行初始化任何的绘图操作。这个延迟给你一个机会去使多个视图内容失效，从视图层级中添加或者移除视图，隐藏视图，调整视图大小以及调整视图位置。你对视图进行所有修改都会在同一时间进行调整。

注意：修改视图的几何形状并不会自动触发系统对视图内容的重绘。视图的 contentMode 属性决定了如何解释视图的几何形状修改。大多数 contentMode 在延伸和重定位已经存在快照的边界，而不是创建一个新的快照。获取更多 contentMode 是怎么影响你的视图绘制周期的，可以查看 [Content Modes](https://developer.apple.com/library/archive/documentation/WindowsViews/Conceptual/ViewPG_iPhoneOS/WindowsandViews/WindowsandViews.html#//apple_ref/doc/uid/TP40009503-CH2-SW2)。

当渲染视图内容时，实际的绘制过程的变化会依赖于视图及其配置。系统视图一般通过实现私有绘图方法来完成视图内容的渲染。这些相同的系统视图经常会暴露出接口供我们配置视图的实际外观。自定义 UIView 类型的子类，通常你要针对你的视图去重载 drawRect: 方法去绘制你视图的内容。这还有一些其它方法提供完成内容的绘制，例如直接设置底层的内容，但是重载 drawRect 方法是目前最常用的方法。

获取更多关于如何绘制自定义视图的内容，可见 [Implementing Your Drawing Code](https://developer.apple.com/library/archive/documentation/WindowsViews/Conceptual/ViewPG_iPhoneOS/CreatingViews/CreatingViews.html#//apple_ref/doc/uid/TP40009503-CH5-SW3)

### 内容模式（content modes）

每个视图都有一个内容模式，其控制了如何根据视图的几何尺寸发生变化时回收视图内容，以及是否回收视图内容。当视图第一次显示时，按照流程绘制其内容并捕获其内容于底层位图中。接下来改变视图几何尺寸并不会导致位图的重新创建。反而，contentMode 属性目的在于位图是否缩放以进行适配新的界限，还是仅简单的贴在一个视图的角或者边上。

视图的内容模式在你做以下操作时生效：

* 改变视图的 frame 或 bounds 矩形区域的宽或高。
* 赋值带有缩放功能的 transform 变换给视图的 transform 属性。

默认情况下大多数视图的 contentMode 属性是 UIViewContentModeScaleToFill，它能够让视图内容缩放适配至新的 frame 大小。图 1-2 展示了一些可用的内容模式结果。从图中可以看到，不是所有的内容模式都会完全充满视图界限，并且有一些还会导致视图内容的失真。

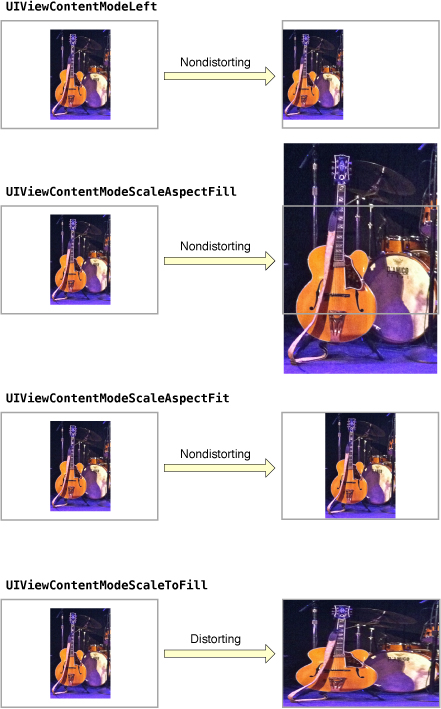


图1-2 不同内容模式下的图片显示

内容模式对于视图回收非常适合，但如果你特别想在缩放或者重设尺寸操作时自定义视图重绘它，还可以使用 UIViewContentModeRedraw 该内容模式。给视图模式设置为该值能够当几何尺寸改变时强制让系统调用 drawRect: 方法。一般情况下，应该尽可能的避免在任何时候使用该值，除非你非常确定你不会使用标志系统视图。

查看更多关于可用的内容模型内容，可看 [UIView Class Reference](https://developer.apple.com/documentation/uikit/uiview)。

### 可伸缩的视图

你可以指定视图的一部分作为可伸缩区域，以至于当视图改变其大小时只修改了可伸缩区域。通常会在按钮或者其它视图中使用可伸缩区域，其中视图的部分区域定义了一个可重复模式。视图中指定的可伸缩区域允许在单独一个轴或者两个轴上进行伸缩。需要注意的是，当对两个轴进行伸缩，对视图的边来说还要定义一个重复模式避免任何变形。图 1-3 清晰的展示了失真是怎么在视图中展示出来的。来自原图中每一个像素点的颜色重复的通过一致的排列充满了大图。

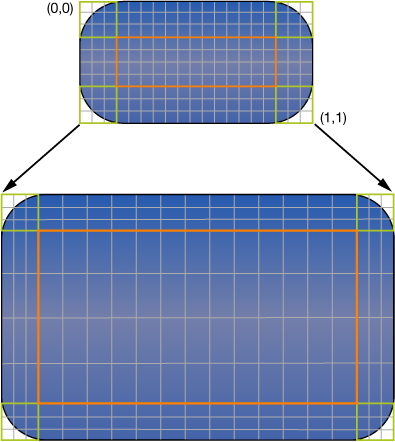


图1-3 内容伸缩示例

使用 contentStretch 属性指定视图的可伸缩区域。这个属性接受一个矩形区域，该值是 0.0 到 1.0 范围内的标准值。当伸缩视图时，系统将会对视图当前的 bounds 值和缩放值与该标准值进行乘积，以达到确定对视图一个像素或多个像素的伸缩。使用标准值能够降低当视图界限改变时更新 contentStretch 属性值的次数。

视图的内容模式还扮演了如何使用视图的伸缩区域决定者的角色。当内容模式导致内容区域的缩放时才会使用可伸缩的区域。这意味着可伸缩视图只支持 UIViewContentModeScaleToFill、UIViewContentModeScaleAspectFit 和 UIViewContentModeScaleAspectFill 内容模式。如果你指定了内容模式对内容进行了贴边或贴角（因此实际上导致内容并不能缩放），该视图将会忽略可伸缩区域。

注意：指定背景视图时，建议创建可伸缩的 UIImage 对象时使用 contentStretch 属性。可伸缩视图可以在 Core Animation Layer 中完全处理，其通常提供更好的性能。

### 内置动画支持

每一个视图后都有一个层对象的好处之一是你可以对与视图相关的更改进行动画化。动画是向用户传递信息交流的有效方法，在设计应用程时应该一直思考这个问题。许多 UIView 的属性都是可动画化的，也就是说支持半自动的从一个值到另外一个值。提高这些其中一个可动画化属性的性能，我们只需要做：

1. 告诉 UIKit 你想要一个高性能动画。
2. 改变属性值。

你可以对 UIView 对象中执行动画的属性如下：

* frame -- 使用这个属性能够对视图位置和大小的改变进行动画变化。
* bounds -- 使用这个属性能对视图大小的改变进行动画变化。
* center -- 使用这个属性能对视图位置的改变进行动画变化。
* transform -- 使用这个属性能够对视图选择和缩放。
* alpha -- 使用这个属性能够修改视图进行透明度的。
* backgroundColor -- 使用这个属性能够修改视图的背景颜色。
* contentStrech -- 使用这个属性能够修改视图的中心延伸。

从一组视图变换到另外一组视图是动画非常重要的一点。通常情况下会使用视图控制器通过动画去管理用户界面各部分中主要修改的部分。举个例子，展示从高级别到低级别视图的过渡信息，通常情况下会使用导航控制器管理每一个成功显示的视图数据。甚至你还可以创建两个视图集合之间的动画过渡来替代视图控制器。在使用标准控制器动画达不到你想要的结果时，可以通过这个方法做到。

除了使用 UIKit 进行动画的创建外，还可以使用 Core Animation layer 完成。下降到图层级别能够对动画的时间和属性进行更多的控制。

查看更多关于如果提升基于视图的动画性能可看 [Animations](https://developer.apple.com/library/archive/documentation/WindowsViews/Conceptual/ViewPG_iPhoneOS/AnimatingViews/AnimatingViews.html#//apple_ref/doc/uid/TP40009503-CH6-SW1)。查看更多关于如何使用 Core Animation 创建动画可看[Core Animation Programming Guide](https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CoreAnimation_guide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40004514) 和 [Core Animation Cookbook.](https://developer.apple.com/library/archive/documentation/GraphicsImaging/Conceptual/CoreAnimation_Cookbook/Introduction/Introduction.html#//apple_ref/doc/uid/TP40005406)。

### 视图的几何形状和定位系统

UIKit 中默认的坐标系统的原点在左上角，并且坐标系从原点向下和向右进行延伸。坐标值使用浮点数表示，它能够进行精确的约束且可让定位的内容不管其下的屏幕分辨率约束。图 1-4 展示了这个相对于屏幕的坐标系统。除了屏幕坐标系统，窗体和视图都定义了它们自己的本地坐标系统，本地坐标系统允许你指定相对与源窗体或源视图的坐标来替换掉相对与屏幕的坐标。

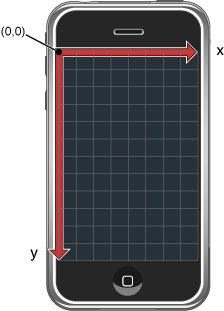


图1-4 iOS系统坐标

**附件4**

**外文文献原文**

# **About Windows and Views**

**Important:** This document is no longer being updated. For the latest information about Apple SDKs, visit the [documentation website](https://developer.apple.com/documentation).

In iOS, you use windows and views to present your application’s content on the screen. Windows do not have any visible content themselves but provide a basic container for your application’s views. Views define a portion of a window that you want to fill with some content. For example, you might have views that display images, text, shapes, or some combination thereof. You can also use views to organize and manage other views.

## At a Glance

Every application has at least one window and one view for presenting its content. UIKit and other system frameworks provide predefined views that you can use to present your content. These views range from simple buttons and text labels to more complex views such as table views, picker views, and scroll views. In places where the predefined views do not provide what you need, you can also define custom views and manage the drawing and event handling yourself.

### Views Manage Your Application’s Visual Content

A view is an instance of the [UIView](https://developer.apple.com/documentation/uikit/uiview) class (or one of its subclasses) and manages a rectangular area in your application window. Views are responsible for drawing content, handling multitouch events, and managing the layout of any subviews. Drawing involves using graphics technologies such as Core Graphics, OpenGL ES, or UIKit to draw shapes, images, and text inside a view’s rectangular area. A view responds to touch events in its rectangular area either by using gesture recognizers or by handling touch events directly. In the view hierarchy, parent views are responsible for positioning and sizing their child views and can do so dynamically. This ability to modify child views dynamically lets your views adjust to changing conditions, such as interface rotations and animations.

You can think of views as building blocks that you use to construct your user interface. Rather than use one view to present all of your content, you often use several views to build a view hierarchy. Each view in the hierarchy presents a particular portion of your user interface and is generally optimized for a specific type of content. For example, UIKit has views specifically for presenting images, text and other types of content.

**Relevant Chapters:** [View and Window Architecture](https://developer.apple.com/library/archive/documentation/WindowsViews/Conceptual/ViewPG_iPhoneOS/WindowsandViews/WindowsandViews.html#//apple_ref/doc/uid/TP40009503-CH2-SW1), [Views](https://developer.apple.com/library/archive/documentation/WindowsViews/Conceptual/ViewPG_iPhoneOS/CreatingViews/CreatingViews.html#//apple_ref/doc/uid/TP40009503-CH5-SW1)

### Windows Coordinate the Display of Your Views

A window is an instance of the [UIWindow](https://developer.apple.com/documentation/uikit/uiwindow) class and handles the overall presentation of your application’s user interface. Windows work with views (and their owning view controllers) to manage interactions with, and changes to, the visible view hierarchy. For the most part, your application’s window never changes. After the window is created, it stays the same and only the views displayed by it change. Every application has at least one window that displays the application’s user interface on a device’s main screen. If an external display is connected to the device, applications can create a second window to present content on that screen as well.

**Relevant Chapters:** [Windows](https://developer.apple.com/library/archive/documentation/WindowsViews/Conceptual/ViewPG_iPhoneOS/CreatingWindows/CreatingWindows.html#//apple_ref/doc/uid/TP40009503-CH4-SW1)

### Animations Provide the User with Visible Feedback for Interface Changes

Animations provide users with visible feedback about changes to your view hierarchy. The system defines standard animations for presenting modal views and transitioning between different groups of views. However, many attributes of a view can also be animated directly. For example, through animation you can change the transparency of a view, its position on the screen, its size, its background color, or other attributes. And if you work directly with the view’s underlying Core Animation layer object, you can perform many other animations as well.

**Relevant Chapters:** [Animations](https://developer.apple.com/library/archive/documentation/WindowsViews/Conceptual/ViewPG_iPhoneOS/AnimatingViews/AnimatingViews.html#//apple_ref/doc/uid/TP40009503-CH6-SW1)

### The Role of Interface Builder

Interface Builder is an application that you use to graphically construct and configure your application’s windows and views. Using Interface Builder, you assemble your views and place them in a nib file, which is a resource file that stores a freeze-dried version of your views and other objects. When you load a nib file at runtime, the objects inside it are reconstituted into actual objects that your code can then manipulate programmatically.

Interface Builder greatly simplifies the work you have to do in creating your application’s user interface. Because support for Interface Builder and nib files is incorporated throughout iOS, little effort is required to incorporate nib files into your application’s design.

For more information about how to use Interface Builder, see [*Interface Builder User Guide*](https://developer.apple.com/library/archive/documentation/DeveloperTools/Conceptual/IB_UserGuide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40005344). For information about how view controllers manage the nib files containing their views, see Creating Custom Content View Controllers in [*View Controller Programming Guide for iOS*](https://developer.apple.com/library/archive/featuredarticles/ViewControllerPGforiPhoneOS/index.html#//apple_ref/doc/uid/TP40007457).

## See Also

Because views are very sophisticated and flexible objects, it would be impossible to cover all of their behaviors in one document. However, other documents are available to help you learn about other aspects of managing views and your user interface as a whole.

* View controllers are an important part of managing your application’s views. A view controller presides over all of the views in a single view hierarchy and facilitates the presentation of those views on the screen. For more information about view controllers and the role they play, see [*View Controller Programming Guide for iOS*](https://developer.apple.com/library/archive/featuredarticles/ViewControllerPGforiPhoneOS/index.html#//apple_ref/doc/uid/TP40007457).
* Views are the key recipients of gesture and touch events in your application. For more information about using gesture recognizers and handling touch events directly, see Event Handling Guide for iOS.
* Custom views must use the available drawing technologies to render their content. For information about using these technologies to draw within your views, see [*Drawing and Printing Guide for iOS*](https://developer.apple.com/library/archive/documentation/2DDrawing/Conceptual/DrawingPrintingiOS/Introduction/Introduction.html#//apple_ref/doc/uid/TP40010156).
* In places where the standard view animations are not sufficient, you can use Core Animation. For information about implementing animations using Core Animation, see [*Core Animation Programming Guide*](https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CoreAnimation_guide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40004514).

# **View and Window Architecture**

Views and windows present your application’s user interface and handle the interactions with that interface. UIKit and other system frameworks provide a number of views that you can use as-is with little or no modification. You can also define custom views for places where you need to present content differently than the standard views allow.

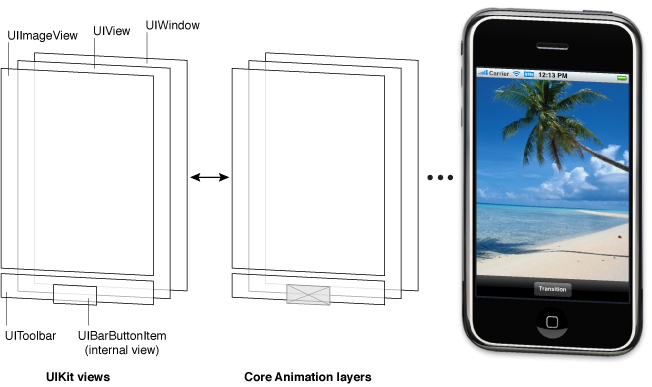
Whether you use the system views or create your own custom views, you need to understand the infrastructure provided by the [UIView](https://developer.apple.com/documentation/uikit/uiview) and [UIWindow](https://developer.apple.com/documentation/uikit/uiwindow) classes. These classes provide sophisticated facilities for managing the layout and presentation of views. Understanding how those facilities work is important for making sure your views behave appropriately when changes occur in your application.

## View Architecture Fundamentals

Most of the things you might want to do visually are done with view objects—instances of the [UIView](https://developer.apple.com/documentation/uikit/uiview) class. A view object defines a rectangular region on the screen and handles the drawing and touch events in that region. A view can also act as a parent for other views and coordinate the placement and sizing of those views. The UIView class does most of the work in managing these relationships between views, but you can also customize the default behavior as needed.

Views work in conjunction with Core Animation layers to handle the rendering and animating of a view’s content. Every view in UIKit is backed by a layer object (usually an instance of the [CALayer](https://developer.apple.com/documentation/quartzcore/calayer) class), which manages the backing store for the view and handles view-related animations. Most operations you perform should be through the UIView interface. However, in situations where you need more control over the rendering or animation behavior of your view, you can perform operations through its layer instead.

To understand the relationship between views and layers, it helps to look at an example. Figure 1-1 shows the view architecture from the [*ViewTransitions*](https://developer.apple.com/library/archive/samplecode/ViewTransitions/Introduction/Intro.html#//apple_ref/doc/uid/DTS40007411) sample application along with the relationship to the underlying Core Animation layers. The views in the application include a window (which is also a view), a generic UIView object that acts as a container view, an image view, a toolbar for displaying controls, and a bar button item (which is not a view itself but which manages a view internally). (The actual [*ViewTransitions*](https://developer.apple.com/library/archive/samplecode/ViewTransitions/Introduction/Intro.html#//apple_ref/doc/uid/DTS40007411) sample application includes an additional image view that is used to implement transitions. For simplicity, and because that view is usually hidden, it is not included in Figure 1-1.) Every view has a corresponding layer object that can be accessed from that view’s [layer](https://developer.apple.com/documentation/uikit/uiview/1622436-layer) property. (Because a bar button item is not a view, you cannot access its layer directly.) Behind those layer objects are Core Animation rendering objects and ultimately the hardware buffers used to manage the actual bits on the screen.



**Figure 1-1**  Architecture of the views in a sample application

The use of Core Animation layer objects has important implications for performance. The actual drawing code of a view object is called as little as possible, and when the code is called, the results are cached by Core Animation and reused as much as possible later. Reusing already-rendered content eliminates the expensive drawing cycle usually needed to update views. Reuse of this content is especially important during animations, where the existing content can be manipulated. Such reuse is much less expensive than creating new content.

### View Hierarchies and Subview Management

In addition to providing its own content, a view can act as a container for other views. When one view contains another, a parent-child relationship is created between the two views. The child view in the relationship is known as the subview and the parent view is known as the superview. The creation of this type of relationship has implications for both the visual appearance of your application and the application’s behavior.

Visually, the content of a subview obscures all or part of the content of its parent view. If the subview is totally opaque, then the area occupied by the subview completely obscures the corresponding area of the parent. If the subview is partially transparent, the content from the two views is blended together prior to being displayed on the screen. Each superview stores its subviews in an ordered array and the order in that array also affects the visibility of each subview. If two sibling subviews overlap each other, the one that was added last (or was moved to the end of the subview array) appears on top of the other.

The superview-subview relationship also impacts several view behaviors. Changing the size of a parent view has a ripple effect that can cause the size and position of any subviews to change too. When you change the size of a parent view, you can control the resizing behavior of each subview by configuring the view appropriately. Other changes that affect subviews include hiding a superview, changing a superview’s alpha (transparency), or applying a mathematical transform to a superview’s coordinate system.

The arrangement of views in a view hierarchy also determines how your application responds to events. When a touch occurs inside a specific view, the system sends an event object with the touch information directly to that view for handling. However, if the view does not handle a particular touch event, it can pass the event object along to its superview. If the superview does not handle the event, it passes the event object to its superview, and so on up the responder chain. Specific views can also pass the event object to an intervening responder object, such as a view controller. If no object handles the event, it eventually reaches the application object, which generally discards it.

For more information about how to create view hierarchies, see [Creating and Managing a View Hierarchy](https://developer.apple.com/library/archive/documentation/WindowsViews/Conceptual/ViewPG_iPhoneOS/CreatingViews/CreatingViews.html#//apple_ref/doc/uid/TP40009503-CH5-SW47).

### The View Drawing Cycle

The [UIView](https://developer.apple.com/documentation/uikit/uiview) class uses an on-demand drawing model for presenting content. When a view first appears on the screen, the system asks it to draw its content. The system captures a snapshot of this content and uses that snapshot as the view’s visual representation. If you never change the view’s content, the view’s drawing code may never be called again. The snapshot image is reused for most operations involving the view. If you do change the content, you notify the system that the view has changed. The view then repeats the process of drawing the view and capturing a snapshot of the new results.

When the contents of your view change, you do not redraw those changes directly. Instead, you invalidate the view using either the [setNeedsDisplay](https://developer.apple.com/documentation/uikit/uiview/1622437-setneedsdisplay) or [setNeedsDisplayInRect:](https://developer.apple.com/documentation/uikit/uiview/1622587-setneedsdisplayinrect) method. These methods tell the system that the contents of the view changed and need to be redrawn at the next opportunity. The system waits until the end of the current run loop before initiating any drawing operations. This delay gives you a chance to invalidate multiple views, add or remove views from your hierarchy, hide views, resize views, and reposition views all at once. All of the changes you make are then reflected at the same time.

**Note:** Changing a view’s geometry does not automatically cause the system to redraw the view’s content. The view’s [contentMode](https://developer.apple.com/documentation/uikit/uiview/1622619-contentmode) property determines how changes to the view’s geometry are interpreted. Most content modes stretch or reposition the existing snapshot within the view’s boundaries and do not create a new one. For more information about how content modes affect the drawing cycle of your view, see [Content Modes](https://developer.apple.com/library/archive/documentation/WindowsViews/Conceptual/ViewPG_iPhoneOS/WindowsandViews/WindowsandViews.html#//apple_ref/doc/uid/TP40009503-CH2-SW2).

When the time comes to render your view’s content, the actual drawing process varies depending on the view and its configuration. System views typically implement private drawing methods to render their content. Those same system views often expose interfaces that you can use to configure the view’s actual appearance. For custom UIView subclasses, you typically override the [drawRect:](https://developer.apple.com/documentation/uikit/uiview/1622529-draw) method of your view and use that method to draw your view’s content. There are also other ways to provide a view’s content, such as setting the contents of the underlying layer directly, but overriding the drawRect: method is the most common technique.

For more information about how to draw content for custom views, see [Implementing Your Drawing Code](https://developer.apple.com/library/archive/documentation/WindowsViews/Conceptual/ViewPG_iPhoneOS/CreatingViews/CreatingViews.html#//apple_ref/doc/uid/TP40009503-CH5-SW3).

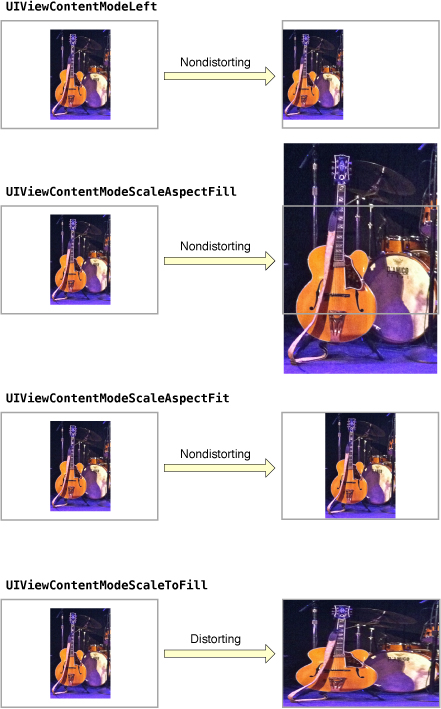
### Content Modes

Each view has a content mode that controls how the view recycles its content in response to changes in the view’s geometry and whether it recycles its content at all. When a view is first displayed, it renders its content as usual and the results are captured in an underlying bitmap. After that, changes to the view’s geometry do not always cause the bitmap to be recreated. Instead, the value in the [contentMode](https://developer.apple.com/documentation/uikit/uiview/1622619-contentmode) property determines whether the bitmap should be scaled to fit the new bounds or simply pinned to one corner or edge of the view.

The content mode of a view is applied whenever you do the following:

* Change the width or height of the view’s [frame](https://developer.apple.com/documentation/uikit/uiview/1622621-frame) or [bounds](https://developer.apple.com/documentation/uikit/uiview/1622580-bounds) rectangles.
* Assign a transform that includes a scaling factor to the view’s [transform](https://developer.apple.com/documentation/uikit/uiview/1622459-transform) property.

By default, the contentMode property for most views is set to [UIViewContentModeScaleToFill](https://developer.apple.com/documentation/uikit/uiviewcontentmode/uiviewcontentmodescaletofill), which causes the view’s contents to be scaled to fit the new frame size. Figure 1-2 shows the results that occur for some content modes that are available. As you can see from the figure, not all content modes result in the view’s bounds being filled entirely, and those that do might distort the view’s content.



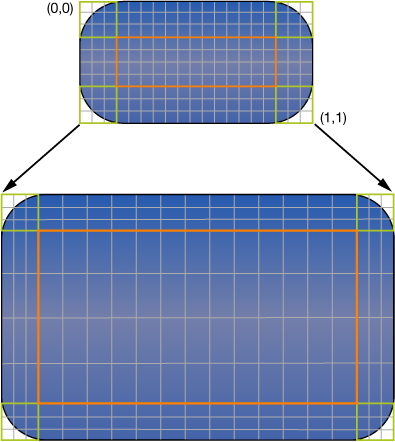
**Figure 1-2**  Content mode comparisons

Content modes are good for recycling the contents of your view, but you can also set the content mode to the  [UIViewContentModeRedraw](https://developer.apple.com/documentation/uikit/uiview/contentmode/redraw) value when you specifically want your custom views to redraw themselves during scaling and resizing operations. Setting your view’s content mode to this value forces the system to call your view’s drawRect: method in response to geometry changes. In general, you should avoid using this value whenever possible, and you should certainly not use it with the standard system views.

For more information about the available content modes, see [*UIView Class Reference*](https://developer.apple.com/documentation/uikit/uiview).

### Stretchable Views

You can designate a portion of a view as stretchable so that when the size of the view changes only the content in the stretchable portion is affected. You typically use stretchable areas for buttons or other views where part of the view defines a repeatable pattern. The stretchable area you specify can allow for stretching along one or both axes of the view. Of course, when stretching a view along two axes, the edges of the view must also define a repeatable pattern to avoid any distortion. Figure 1-3 shows how this distortion manifests itself in a view. The color from each of the view’s original pixels is replicated to fill the corresponding area in the larger view.



**Figure 1-3**  Stretching the background of a button

You specify the stretchable area of a view using the [contentStretch](https://developer.apple.com/documentation/uikit/uiview/1622511-contentstretch) property. This property accepts a rectangle whose values are normalized to the range 0.0 to 1.0. When stretching the view, the system multiplies these normalized values by the view’s current bounds and scale factor to determine which pixel or pixels need to be stretched. The use of normalized values alleviates the need for you to update the contentStretchproperty every time the bounds of your view change.

The view’s content mode also plays a role in determining how the view’s stretchable area is used. Stretchable areas are only used when the content mode would cause the view’s content to be scaled. This means that stretchable views are supported only with the [UIViewContentModeScaleToFill](https://developer.apple.com/documentation/uikit/uiviewcontentmode/uiviewcontentmodescaletofill), [UIViewContentModeScaleAspectFit](https://developer.apple.com/documentation/uikit/uiviewcontentmode/uiviewcontentmodescaleaspectfit), and [UIViewContentModeScaleAspectFill](https://developer.apple.com/documentation/uikit/uiviewcontentmode/uiviewcontentmodescaleaspectfill) content modes. If you specify a content mode that pins the content to an edge or corner (and thus does not actually scale the content), the view ignores the stretchable area.

**Note:** The use of the contentStretch property is recommended over the creation of a stretchable UIImage object when specifying the background for a view. Stretchable views are handled entirely in the Core Animation layer, which typically offers better performance.

### Built-In Animation Support

One of the benefits of having a layer object behind every view is that you can animate many view-related changes easily. Animations are a useful way to communicate information to the user and should always be considered during the design of your application. Many properties of the [UIView](https://developer.apple.com/documentation/uikit/uiview)class are animatable—that is, semiautomatic support exists for animating from one value to another. To perform an animation for one of these animatable properties, all you have to do is:

1. Tell UIKit that you want to perform an animation.
2. Change the value of the property.

Among the properties you can animate on a UIView object are the following:

* [frame](https://developer.apple.com/documentation/uikit/uiview/1622621-frame)—Use this to animate position and size changes for the view.
* [bounds](https://developer.apple.com/documentation/uikit/uiview/1622580-bounds)—Use this to animate changes to the size of the view.
* [center](https://developer.apple.com/documentation/uikit/uiview/1622627-center)—Use this to animate the position of the view.
* [transform](https://developer.apple.com/documentation/uikit/uiview/1622459-transform)—Use this to rotate or scale the view.
* [alpha](https://developer.apple.com/documentation/uikit/uiview/1622417-alpha)—Use this to change the transparency of the view.
* [backgroundColor](https://developer.apple.com/documentation/uikit/uiview/1622591-backgroundcolor)—Use this to change the background color of the view.
* [contentStretch](https://developer.apple.com/documentation/uikit/uiview/1622511-contentstretch)—Use this to change how the view’s contents stretch.

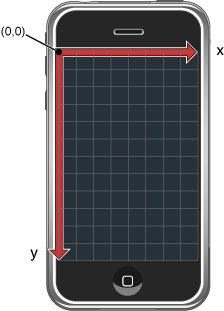
One place where animations are very important is when transitioning from one set of views to another. Typically, you use a view controller to manage the animations associated with major changes between parts of your user interface. For example, for interfaces that involve navigating from higher-level to lower-level information, you typically use a navigation controller to manage the transitions between the views displaying each successive level of data. However, you can also create transitions between two sets of views using animations instead of a view controller. You might do so in places where the standard view-controller animations do not yield the results you want.

In addition to the animations you create using UIKit classes, you can also create animations using Core Animation layers. Dropping down to the layer level gives you much more control over the timing and properties of your animations.

For details about how to perform view-based animations, see [Animations](https://developer.apple.com/library/archive/documentation/WindowsViews/Conceptual/ViewPG_iPhoneOS/AnimatingViews/AnimatingViews.html#//apple_ref/doc/uid/TP40009503-CH6-SW1). For more information about creating animations using Core Animation, see [*Core Animation Programming Guide*](https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CoreAnimation_guide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40004514) and [*Core Animation Cookbook*](https://developer.apple.com/library/archive/documentation/GraphicsImaging/Conceptual/CoreAnimation_Cookbook/Introduction/Introduction.html#//apple_ref/doc/uid/TP40005406).

## View Geometry and Coordinate Systems

The default coordinate system in UIKit has its origin in the top-left corner and has axes that extend down and to the right from the origin point. Coordinate values are represented using floating-point numbers, which allow for precise layout and positioning of content regardless of the underlying screen resolution. Figure 1-4 shows this coordinate system relative to the screen. In addition to the screen coordinate system, windows and views define their own local coordinate systems that allow you to specify coordinates relative to the view or window origin instead of relative to the screen.



**Figure 1-4**  Coordinate system orientation in UIKit