第 2 章

设计服务架构

本章内容：

- 实现远程门面
- 使用服务定位器探测端点
- 使用服务版本化支持老版本应用

本章代码下载

可以在 www.wrox.com 的 Download Code 选项卡中下载本章代码，网址是 www.wrox.com/WileyCDA/WroxTitle/Professional-iOS-Network-Programming-Connecting-the-Enterprise-to-the-iPhone-and-iPad.productCd-1118362403.html。本章下载代码中包含一个示例项目与一套 Web Serive：

- Facade Tester.zip
- Facade PHP.zip

Web Service 是 iOS 网络应用的根本所在，其设计的灵活性与健壮性会对用户体验产生极大的影响。设计良好的服务 API 可以适应变化的后端数据源，同时为依赖它的应用提供不变的门面。服务定位器可以使应用能够动态探测到新的服务端点并使用它们，从而无须重新编译并重新向 App Store 提交应用。如果有必要重新提交应用，那么需要在过渡与升级期间支持应用的老版本，这是应用整个生命周期的一部分。如果想支持依旧被每天使用的老版本应用，同时又不妨碍为新版本增加新的特性，那么支持版本化的服务 API 就显得尤为重要了。本章将会在真实的业务场景中介绍这些重要的设计元素，同时会给出相应的示例实现。

2.1　远程门面模式

在设计应用的服务架构时，远程门面可以简化应用集成，并且可以让多个客户端共享相同的业务逻辑。门面模式用于抽象出客户端所用的底层系统的复杂性。比如，邮政系统包括大量的邮递员、卡车、飞机、配送中心和邮局；然而，用户所需的大多数任务都将这种复杂性隐藏起来了，只包含邮寄信件与接收包裹。用户无须了解信件是如何从纽约到达旧金山的，他们只需要支付邮资，然后等待邮件的到达即可。与之类似，应用 API 也可以将多个数据库查询或是后端系统请求抽象为一个单独的外部访问方法，该方法会返回操作的结果。只要门面的外部 API 契约保持不变，底层系统可以变化、升级或是完全移除而不会对使用该门面的任何客户端造成影响。

远程门面也使用了这种模式，并且将其用在了应用的 Web Service 层。它定义了一个不变的服务契约，应用可以通过该契约在外部创建、读取、更新或删除数据。API 通常用于与现有的业务系统进行交互，并提供具有相同功能的移动版本。图 2-1 展示了应用是如何直接查询各种端点的，图 2-2 展示了当门面与代表应用的各种后端服务进行交互时网络拓扑的变化情况。如果在设计最初的服务契约时能够保持谨慎并有预见性，那么相同的 API 就可以适应后端系统的大多数变化，这样应用就无须频繁更新来匹配服务基础设施，从而不会对应用的功能造成影响。

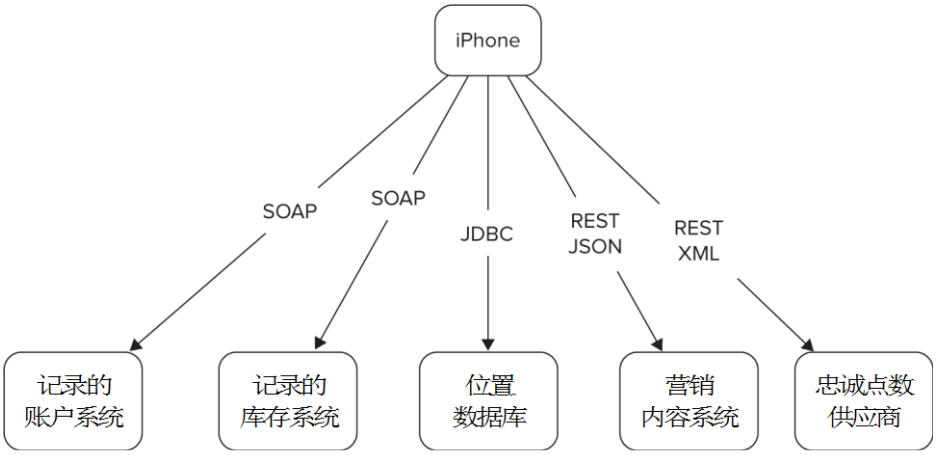

图 2-1

假设有家银行与其竞争对手合并了，该银行想将其现有的账户迁移到竞争对手的账户存储系统中。如果服务 API 的编写采用了抽象的银行功能，那么它就可以处理提供相同数据的任何后端数据库，即便存储在新的格式中也是如此。远程门面可以切换到新的数据源，转换与 API 契约不匹配的任何数据，然后将其返回给移动银行应用，而用户则完全不知道后面发生了什么事情。这种开发方式叫做契约编程，它能确保网络会话的两端遵循之前达成一致的输入与输出契约。只要契约依然有效，那么后端系统无论是重写、移植到其他语言还是升级，都不会对另一端造成任何负面影响。

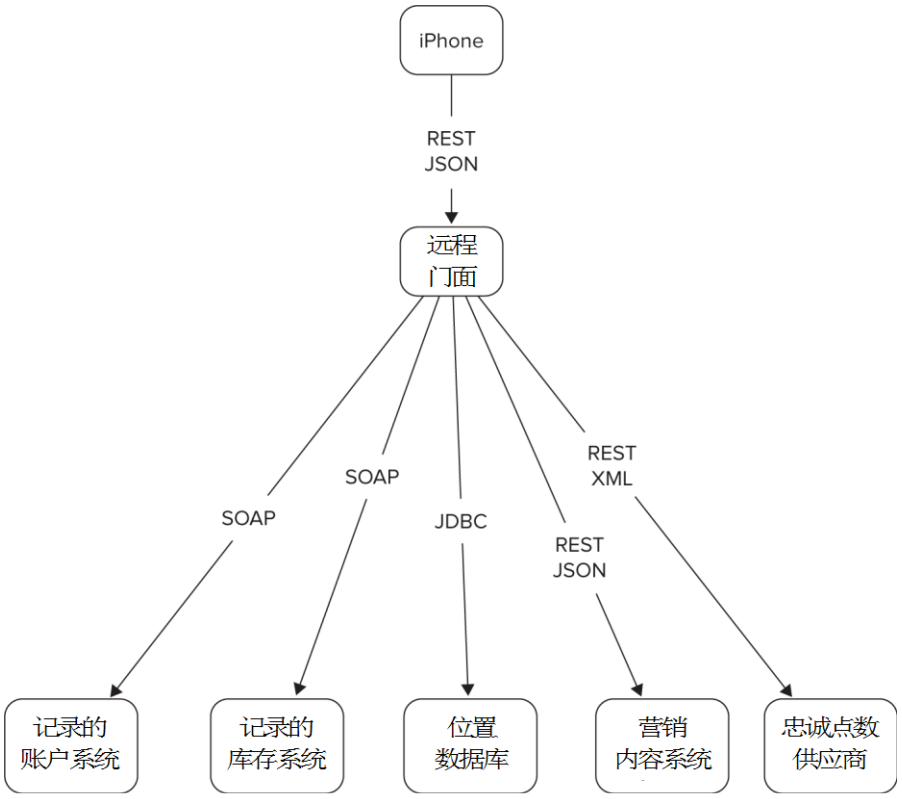

图 2-2

契约中应用一方的可维护性、可靠性与复杂性都会因门面模式而得到极大改善。随着应用中网络交互点数量的下降，需要支持未来门面版本的变化数量也会变得越来越少，它可以是自包含的。可靠性之所以会得到改进，是因为门面通常只有一个协议和一种消息格式，这样就降低了针对其他格式的第三方库或是单独的解析器的数量。这些变化都会降低应用的复杂性，节省开发成本，因为涵盖所有功能的单元测试数量会变少。在服务器端，我们只需要保护和向 Internet 公开一套端点即可，而不再是向众多不同的系统公开。

远程门面还会促使开发者将应用中的一些业务逻辑放到服务层中。某些变化频繁或是无法提前预估的函数可以在服务层中进行计算，然后只将最后的结果发送给客户端。在这种方式下，如果需要根据新的业务规则调整逻辑，那么应用无须更新即可生效。在银行合并这个示例中，这种调整可能是新的机构所采用的新的密码安全需求。如果应用只接收用户的密码，然后询问门面密码是否合法，那么逻辑就可以随意变化了。采用类似的模式来验证 e-mail 地址则可以轻松适应即将到来的自定义顶层域名(Top-Level Domain，TLD)；然而，如果有效的 TLD 列表是在应用中硬编码的，可能就会拒绝掉合法的 E-mail 地址，直到发布应用更新为止。在面对变化的业务流程时，远程门面可以在网络应用发布后确保企业拥有最大的灵活性。

相同的特性也适用于 API 的输入。门面可以将请求转换为后端系统所需的格式；比如，可以将 JSON 请求转换为 SOAP 请求，还可以对无法公开到 Internet 的其他系统强制施加

安全约束，在转发请求前追踪和验证 API key，或是限制发送给某些后端系统的请求数量。

2.1.1　门面服务示例

Facade Tester 应用使用两个 Web Service 装配视图：股票报价服务和天气服务。这两个服务从两个独立的源获取各自的数据，然后将数据转换为一种常见的输出格式。这么做模拟了一种门面服务，当应用在运行时，服务必须能在两个后端系统之间进行切换。这两个示例使用的都是版本 1 服务；版本 2 服务则用在 2.2 节“服务版本化”中。

股票报价服务将数据加载为逗号分隔的值(Comma-Separated Value，CSV)或是 XML 文档，如代码清单 2-1 所示。

代码清单 2-1　从两个股票报价源生成常见的输出(stockQuote_v1.php)

```
<?php
$useYahooResults = true;
$ticker = "AAPL";

if($useYahooResults) {
    $rawData = rtrim(file_get_contents("http://finance.yahoo.com/ d/
        quotes.csv?s=".$ticker."&f=snl1p2o"),"\r\n");

    $data = explode(",",$rawData);

    $symbol = trim($data[0],'"');
    $name = trim($data[1],'"');
    $currentPrice = trim($data[2],'"');

} else {
    $rawXML = file_get_contents("http://www.webservicex.net/stockquote.asmx/
        GetQuote?symbol=".$ticker);

        $wrapperData = simplexml_load_string($rawXML);

        $xmlData = simplexml_load_string($wrapperData);

        $symbol = (string)$xmlData->Stock->Symbol;
        $name = (string)$xmlData->Stock->Name;
        $currentPrice = (string)$xmlData->Stock->Last;
    }

    $response = array("symbol" => $symbol,
                      "name" => $name,
                      "currentPrice" => $currentPrice);

    // output final results:
    printjson_encode($response);

    ?>
```

下述逗号分隔的字符串具有关键的股票值：公司名、最近行市、开盘价以及开盘后的变动百分比。

```
{ticker symbol},{name},{last trade price},{percentage change},{opening price}

example:
"AAPL","Apple Inc.",530.12,"-2.92%",545.31
```

下述 XML 文档以更加结构化的格式展现出了相同的数据。该文档包含了 CSV 字符串中的所有信息，另外又加上了该服务忽略掉的一些额外数据。

```
<StockQuotes>
   <Stock>
      <Symbol>AAPL</Symbol>
      <Last>530.12</Last>
      <Date>5/17/2012</Date>
      <Time>4:00pm</Time>
      <Change>-15.955</Change>
      <Open>545.31</Open>
      <High>547.50</High>
      <Low>530.12</Low>
      <Volume>25614960</Volume>
      <MktCap>495.7B</MktCap>
      <PreviousClose>546.075</PreviousClose>
      <PercentageChange>-2.92%</PercentageChange>
      <AnnRange>310.50 - 644.00</AnnRange>
      <Earns>41.042</Earns>
      <P-E>13.31</P-E>
      <Name>Apple Inc.</Name>
   </Stock>
</StockQuotes>
```

如果变量$useYahooResults 为 true，就会加载 CSV 字符串，否则就会加载 XML。无论输入源是什么，门面都会以常见的 JSON 格式返回其数据，如下所示：

```
{"symbol":"AAPL","name":"AppleInc.","currentPrice":"-2.92%"}
```

门面所用的任何数据源都至少要提供所需的最低限度的字段，从而遵守它与 API 客户端所达成的契约。

示例门面还实现了一个 Web Service，从两个源之一提供弗吉尼亚州里士满当前的天气。这两个源都以 JSON 形式提供了天气状况，不过每个源具体的响应格式则大相径庭。该服务类似于这种情况：你想要将后端系统升级到新版本，新旧版本具有相同的基础数据，不过数据的组织形式是完全不同的。代码清单 2-2 展示了这个天气服务，它遵循与股票服务相同的基础结构。

代码清单 2-2　从两个天气服务生成常见的输出(weather_v1.php)

```
<?php
$useYahooResults = true;
```

```
if($useYahooResults) {
    $rawJSON = file_get_contents("http://query.yahooapis.com/v1/
                public/yql?q=select%20item%20from%20weather.forecast
                %20where%20location%3D%2248907%22&format=json");
    $rawData = json_decode($rawJSON);

$currentTemperature = $rawData->query->results->channel->item->
                                                    condition->temp;
$currentConditions = $rawData->query->results->channel->item->
                                                    condition->text;

} else {
    $rawJSON = file_get_contents("http://weather.yahooapis.com/forecastjson?
        w=12518996");
    $rawData = json_decode($rawJSON);

    $currentTemperature = (string)$rawData->condition->temperature;
    $currentConditions = $rawData->condition->text;
}

$response = array( "city" => "Richmond",
                   "state" => "Virginia",
                   "currentTemperature" => $currentTemperature);

/*
 * output final results:
 *
 * {"city":"Richmond","state":"Virginia","currentTemperature":"63"}
 */
printjson_encode($response);

?>
```

现在，消费客户端可以使用发布的每个服务，数据源也可以动态切换而无须修改其处理股票或天气数据的方式。

2.1.2　门面客户端示例

Facade Tester 应用说明了如何使用输出格式并在表视图中展现结果。代码清单 2-3 展示了如何通过 Grand Central Dispatch 在后台线程中加载门面天气服务。代码清单 2-4 展示了解析股票报价服务的相关代码。JSON 结果是通过 iOS 5 的 NSJSONSerialization 解析的，然后赋给了表视图所用的局部变量。这证明了门面模式确实能减轻我们的工作量，iOS 集成的关键代码只有区区几行。要想了解关于通过网络加载数据的更多信息，请参见第 3 章“构建请求”；要想了解关于 JSON 解析的更多信息，请参见第 4 章“生成与解析负载”。

代码清单 2-3　加载并解析天气服务(FTWeatherViewController.m)

```
NSString *v1_city;
```

```
NSString *v1_state;
NSString *v1_temperature;

- (void)loadVersion1Weather {
    dispatch_async(dispatch_get_global_queue(
        DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{

        FTAppDelegate *appDelegate = (FTAppDelegate*)
            [[UIApplicationsharedApplication] delegate];

        if(appDelegate.urlForWeatherVersion1 != nil) {
            NSError *error = nil;
            NSData *data = [NSData
                   dataWithContentsOfURL:appDelegate.urlForWeatherVersion1
                   options:NSDataReadingUncached
                   error:&error];
            if(error == nil) {
                NSDictionary *weatherDictionary = [NSJSONSerialization
                    JSONObjectWithData:data
                           options:NSJSONReadingMutableLeaves
                             error:&error];
                if(error == nil) {
                    v1_city = [weatherDictionaryobjectForKey:@"city"];
                    v1_state = [weatherDictionaryobjectForKey:@"state"];
                    v1_temperature = [weatherDictionaryobjectForKey:
                        @"currentTemperature"];

                    // update the table on the UI thread
                    dispatch_async(dispatch_get_main_queue(), ^{
                        [self.tableViewreloadData];
                    });

                } else {
                    NSLog(@"Unable to parse weather because of error: %@",
                                                                  error);
                      [selfshowParseError];
                }

            } else {
                [selfshowLoadError];
            }
        } else {
            [selfshowLoadError];
        }
    });
}
```

代码清单 2-4　加载并解析股票报价服务(FTStockViewController.m)

```
NSString *v1_symbol;
NSString *v1_name;
NSNumber *v1_currentPrice;

- (void)loadVersion1Stock {
    dispatch_async(dispatch_get_global_queue(
        DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{

        FTAppDelegate *appDelegate = (FTAppDelegate*)
            [[UIApplicationsharedApplication] delegate];

        if(appDelegate.urlForStockVersion1 != nil) {
            NSError *error = nil;
            NSData *data = [NSDatadataWithContentsOfURL:
                                appDelegate.urlForStockVersion1
                                options:NSDataReadingUncached
                                            error:&error];
            if(error == nil) {
                NSDictionary *stockDictionary = [NSJSONSerialization
                    JSONObjectWithData:data
                              options:NSJSONReadingMutableLeaves
                                error:&error];
                if(error == nil) {
                    v1_symbol = [stockDictionaryobjectForKey:@"symbol"];
                    v1_name = [stockDictionaryobjectForKey:@"name"];
                    v1_currentPrice = [NSNumbernumberWithFloat:
                        [[stockDictionaryobjectForKey:@"currentPrice"]
                        floatValue]];

                    // update the table on the UI thread
                    dispatch_async(dispatch_get_main_queue(), ^{
                        [self.tableViewreloadData];
                    });

                } else {
                    NSLog(@"Unable to parse stock quote because of error:
                        %@", error);
                    [selfshowParseError];
                }
            } else {
                [selfshowLoadError];
            }
        } else {
```

```
            [selfshowLoadError];
        }
    });
}
```

2.2　服务版本化

移动应用会经常更新以修复 Bug 和添加新特性，不过有些人常常觉得 Web Service 也需要维护并随之更新。服务版本化是这样一种技术，它更新与客户端的 API 契约，同时依然保留之前的版本以供现有的应用版本使用。通过 App Store 发布的应用无法强制用户升级到最新版本，这意味着在过渡期间现有的 Web Service 依然要保持功能的可用性。根据用户的升级行为，关闭现有服务而不影响旧版本的用户几乎是不可能的。一种方式就是加入检查逻辑，检查最小支持的应用版本，然后显示升级消息，直到用户同意升级为止。然而，在之前可以使用的版本上显示的这些烦人的消息可能会让一些用户感到心烦意乱，他们可能很快就会打爆你的支持热线，并在 App Store 评论中给出差评。由于这些潜在的负面效果，恰当的服务版本化确确实实是最佳解决方案。

API 版本化并不仅仅限于新的应用升级；它还可以用于向各种类型的设备发布不同的或是扩展的数据。比如，一个报表应用可能在 iPhone 上显示时需要一套数据，但在 iPad 上显示时则需要更加完整的数据集，因为 iPad 拥有更大的屏幕尺寸。如果这些额外的数据有着较大的后端或是网络负载，那么你肯定不想在并不需要它们的 iPhone 服务请求上浪费这些资源。

版本化系统的结构主要有两种方式：一种是主动系统，远程门面会接收到客户端的当前版本，然后选择正确的端点；另一种是被动系统，版本化服务端点是硬编码到客户端的每个新发布中。到底哪一种才是提供版本号作为输入的最佳方式，则是由每一家企业决定的，不过通常情况下，版本号会包含在 REST 端点的 URL 结构中或是作为查询参数进行传递。下述代码片段展示了这两种版本输入方式：

```
// a version given in the URL structure
http://example.com/api/1.0/stockquote/AAPL

// a version given as a query parameter
http://example.com/api/stockQuote.php?ticker=AAPL&apiVersion=1.0
```

被动系统是实现服务版本化最简单的方式。这种方式无须规划额外的服务器，在组织上的代价要大于技术上的。要想实现这种方式，只需要将版本号硬编码到客户端应用中已经定义好的端点 URL 中。由于在应用发布后，这些 URL 在功能上就是不变的了，因此可以确保硬编码使用该版本的应用总是会使用该版本。当新的客户端版本需要改变服务契约时，你只需要增加硬编码的 API 版本号，然后创建新的 Web Service 即可。

主动系统拥有被动系统的全部优势；然而就像所有的门面交互一样，它还具有未来改

变行为的能力。如果相同 Web Service 的两个不同版本具有相同的输入与输出契约时，但它们所执行的某些计算是不同的，这样兼容于两个版本的老客户端就可以在未来动态切换了。比如，如果某个在线零售商现在没有征收某个州的销售税，那么可以将客户端发送至价格检查服务的 1.0 版。然而，如果未来需要开始征收销售税，那么他只需要创建 2.0 版的服务，返回正常价格加上税就可以了。假设两种情况下返回的最终结果都是数字，那么服务契约就不会受到影响。要想实现主动版本化系统，门面需要将所有可能的客户端应用版本组合到兼容性桶(compatibility bucket)当中，然后为每个桶分配正确的 API 版本供其使用。为了简化未来的开发，请为高于门面所知道的最大版本号的客户端版本选择好的默认值，通常是最近的 API 版本。

2.2.1　版本化服务示例

两个示例 Web Service 都有两个版本，模拟了随着业务需求变化而扩展的服务契约。有些输出字段类型已经被修改，还有些新增的字段。这些示例使用了被动版本化系统，仅仅通过修改 URL 来指定新的版本。回忆一下天气服务的 1.0 版 weather_v1.php，它有如下输出格式：

```
{"city":"Richmond","state":"Virginia","currentTemperature":"63"}
```

currentTemperature 表示为字符串，这使得客户端在需要整型逻辑时变得复杂，比如设置用于对当前温度进行分类的寒冷、温和或炎热天气的阀值。服务的 2.0 版修复了这个疏忽，将返回值定义为数值类型。它还添加了 currentConditions 字段，这是对当前天气的文本说明。weather_v2.php 的输出格式如下所示：

```
{"city":"Richmond","state":"Virginia","currentTemperature":63,
 "currentConditions":"Mostly Cloudy"}
```

股票报价服务的 1.0 版与 2.0 版之间的变化也与此类似。stockQuote_v1.php 的第一个版本提供了基本的输出：

```
{"symbol":"AAPL","name":"Apple Inc.","currentPrice":"530.12"}
```

注意 currentPrice 在 1.0 版中是字符串，但在 stockQuote_v2.php 中却表示为数字，这样可以简化其在客户端的格式化：

```
{"symbol":"AAPL","name":"Apple Inc.","openingPrice":545.31,
 "currentPrice": 530.12,"percentageChange":"-2.92%"}
```

此外，还添加了两个新字段：openingPrice 与 percentageChange。

2.2.2　使用版本化服务的客户端示例

Facade Tester 中的天气视图控制器可以显示两个 API 版本的输出。loadVersion1Weather 与 loadVersion2Weather 会检查 API 端点的 URL 的应用委托，如代码清单 2-5 中的粗体代码所示。由于该例使用的是被动版本化，因此在这里直接硬编码 URL 看起来更加自然；然

而，在应用委托中定义可以使得在实现服务定位器时更具灵活性，下一节将会对此进行介绍。

代码清单 2-5　从应用委托中获取 API 端点(FTWeatherViewController.m)

```
- (void)loadVersion1Weather {
    dispatch_async(dispatch_get_global_queue(
        DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{

        FTAppDelegate *appDelegate = (FTAppDelegate*)
            [[UIApplicationsharedApplication] delegate];
        if(appDelegate.urlForWeatherVersion1 != nil) {
            NSError *error = nil;
            NSData *data = [NSData
dataWithContentsOfURL:appDelegate.urlForWeatherVersion1
                            options:NSDataReadingUncached
                            error:&error];

        // remaining code removed for brevity
}

- (void)loadVersion2Weather {
    dispatch_async(dispatch_get_global_queue(
        DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{

        FTAppDelegate *appDelegate = (FTAppDelegate*)
            [[UIApplicationsharedApplication] delegate];

        if(appDelegate.urlForWeatherVersion2 != nil) {
            NSError *error = nil;
            NSData *data = [NSData
dataWithContentsOfURL:appDelegate.urlForWeatherVersion2
                            options:NSDataReadingUncached
                            error:&error];

        // remaining code removed for brevity
}
```

应用在加载正确的 JSON 数据后，只是根据该版本的服务契约对其进行解析。天气服务的 1.0 版会加载城市、州与当前温度，如下所示：

```
v1_city = [weatherDictionaryobjectForKey:@"city"];
v1_state = [weatherDictionaryobjectForKey:@"state"];
v1_temperature = [weatherDictionaryobjectForKey:@"currentTemperature"];
```

2.0 版与之类似，不过会将当前温度解析为数字，还会查找当前的情况，代码如下所示：

```
v2_city = [weatherDictionaryobjectForKey:@"city"];
v2_state = [weatherDictionaryobjectForKey:@"state"];
v2_temperature = [[weatherDictionaryobjectForKey:@"currentTemperature"]
```

```
intValue];
    v2_conditions = [weatherDictionaryobjectForKey:@"currentConditions"];
```

在设置v2_temperature时，将之从NSNumber(NSJSONSerialization所用的数值类型)转换为视图控制器所用的整型。

2.3　服务定位器

服务定位器是一个帮助应用动态探测远程源 API 端点的工具，它可以解决应用硬编码的无效或不再存在的端点问题。应用开发者还可以通过服务定位器将之前发布的应用重新指向可用的新服务。这些新服务无须改变与客户端的 API 契约；比如，端点移到了不同的服务器或子域中、位于负载平衡器之后，或是移到了由 SSL 保证安全的 HTTPS 端点上。甚至可以为每个开发环境创建新的服务定位器文件以便轻松在开发、QA 或生产资源间切换，而这一切只需一次变换即可。

服务定位器的核心只是一个包含了 API 端点与关于这些端点的一些简要元数据的文件。应用通过这些元数据确定该使用哪个端点。比如 API 版本、输入或输出格式、设备类型以及安全级别等。它还需要包含端点的 URL 以及客户端应用用于匹配端点与其函数的键。由于该文件是静态的，不会频繁修改，因此可以轻松将其部署到 Web 服务器或内容分发网络(CDN)上。服务定位器的源需要高度可靠，因为它是应用失败的单点。虽然这看起来有点问题，但如果应用直接查询每个独立的后端服务，就会有很多个失败点存在，相比于此，单点失败会更好一些。在可能的情况下，服务定位器应该是负载平衡的，从而避免全部的用户请求发给一台单独主机。由于 CDN 的设计目的是针对静态文件的高可靠性，通常能比平常的 Web 服务器处理更高的持续带宽占有情况，因此我们推荐你在可能的情况下使用 CDN 来服务于服务定位器文件。

由于大多数 Web Service 都会将结果以 JSON 的形式输出，因此使用 JSON 来表示服务定位器会比较好。代码清单 2-6 展示了 Facade Tester 如何使用服务定位器探测天气与股票报价 API 端点。该结构将端点的所有版本都组合到了一个文件中；然而，你还可以为每个 API 版本创建单独的服务定位器文件。后者可以防止应用版本混合在一起并匹配不同的服务版本，不过对于某些业务场景来说这种约束并不是什么问题。

代码清单 2-6　示例服务定位器文件(serviceLocator.json)

```
{
   "services": [
      {
         "name": "stockQuote",
         "url": "http://example.com/api/stockQuote_v1.php",
         "version": 1
      },
      {
         "name": "stockQuote",
```

```
            "url": "http://example.com/api/stockQuote_v2.php",
            "version": 2
        },
        {
            "name": "weather",
            "url": "http://example.com/api/weather_v1.php",
            "version": 1
        },
        {
            "name": "weather",
            "url": "http://example.com/api/weather_v2.php",
            "version": 2
        }
    ]
}
```

实现了服务定位器模式的任何客户端的第一个动作通常都是加载并解析文件。由于所有的网络调用都需要端点，而端点只位于该文件中，因此在任何其他的网络动作发生之前必须先解析该文件。当应用返回到前台以确保端点数据是最新时，定位器文件也应该进行更新。应用可以停留在后台状态中一段时间，它之前可能已经加载了一个服务定位器文件，不过文件现在可能已经不是最新的了。在某些情况下，过时的端点可能会退出并超时，这会导致差劲的用户体验。通常情况下，当服务定位器加载时，应用会显示启动画面。

代码清单 2-7 展示了一个应用，它会在应用启动和返回到前台时加载服务定位器。它将 URL 存储为应用委托中的属性；然而，更加复杂的应用则需要专门的网络管理器，由它来处理服务定位器的加载，其他控制器也会使用它针对特定的网络调用查询端点。

代码清单 2-7　加载并解析服务定位器文件(FTAppDelegate.m)

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    // some code removed for brevity
    /*
     * load the service locator
     *
     * note: You should probably show a splash screen of some kind here
     * that waits for the SL to fully load. Currently a user could
     * try to start a network request before it knows which URL to use.
     */
    [selfloadServiceLocator];

    return YES;
}

- (void)applicationDidBecomeActive:(UIApplication *)application {
    // load the service locator
```

```
    [selfloadServiceLocator];
}

- (void)loadServiceLocator {
    dispatch_async(dispatch_get_global_queue(
        DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{


        NSError *error = nil;
        NSData *data = [NSDatadataWithContentsOfURL:[NSURL URLWithString:
                        @"http://example.com/api/serviceLocator.json"]
                                            options:NSDataReadingUncached
                                              error:&error];

        if(error == nil) {
            NSDictionary *locatorDictionary = [NSJSONSerialization
                JSONObjectWithData:data
                           options:NSJSONReadingMutableLeaves
                             error:&error];
            if(error == nil) {
                self.urlForStockVersion1 = [self
                    findURLForServiceNamed:@"stockQuote"
                                   version:1
                              inDictionary:locatorDictionary];
                self.urlForStockVersion2 = [self
                    findURLForServiceNamed:@"stockQuote"
                                   version:2
                              inDictionary:locatorDictionary];
                self.urlForWeatherVersion1 = [self
                    findURLForServiceNamed:@"weather"
                                   version:1
                              inDictionary:locatorDictionary];
                self.urlForWeatherVersion2 = [self
                    findURLForServiceNamed:@"weather"
                                   version:2
                              inDictionary:locatorDictionary];

            } else {
                NSLog(@"Unable to parse service locator because of error:
                    %@", error);

                // inform the user on the UI thread
                dispatch_async(dispatch_get_main_queue(), ^{
                    [[[UIAlertViewalloc] initWithTitle:@"Error"
                                               message:@"Unable to parse
```

```
                                                    service locator."
                                            delegate:nil
                                   cancelButtonTitle:@"OK"
                                   otherButtonTitles:nil] show];
            });
        }

    } else {
        NSLog(@"Unable to load service locator because of error: %@", error);

        // inform the user on the UI thread
        dispatch_async(dispatch_get_main_queue(), ^{
            [[[UIAlertViewalloc] initWithTitle:@"Error"
                                       message:@"Unable to load service
                                                locator. Did you remember
                                                to update the URL to your own
                                                copy of it?"
                                      delegate:nil
                             cancelButtonTitle:@"OK"
                             otherButtonTitles:nil] show];
        });
    }
});
}

- (NSURL*)findURLForServiceNamed:(NSString*)serviceName
                         version:(NSInteger)versionNumber
                    inDictionary:(NSDictionary*)locatorDictionary {

    NSArray *services = [locatorDictionaryobjectForKey:@"services"];

    for(NSDictionary *serviceInfo in services) {
        NSString *name = [serviceInfoobjectForKey:@"name"];
        NSInteger version = [[serviceInfoobjectForKey:@"version"] intValue];

        if([name caseInsensitiveCompare:serviceName] == NSOrderedSame&&
            version == versionNumber) {

            return [NSURL URLWithString:[serviceInfoobjectForKey:@"url"]];
        }
    }

    return nil;
}
```

2.4　小结

灵活的服务架构需要在应用的第一个版本发布之前经过精心的规划与实现，这样才能获取最大的收益。如果某个版本采用了硬编码的端点或是业务逻辑，那么可以有效地支持该配置，即便业务发生了巨大变化也是如此。通过远程门面、API 版本化与服务定位器的融合，可以采用多种方式来改变已发布应用的业务逻辑与 API 设置。现在，对产品代码进行细微修改，以及在新版本中增加主要的新特性而无须破坏之前的应用版本已经变得很容易了。之前的服务基础设施的开发代价看起来可能没有必要，不过随着应用逐渐变大和演化，这种代价会带来很多倍的收益。