	原则与特性		结构
√	功能多样性	√	模块
	概念完整性	√	依赖关系
√	修改独立性		进程
√	自动传播	√	数据访问
	可构建性		
√	增长适应性		
√	熵增抵抗力		

第 6 章

数据增长：Facebook平台的架构

Dave Fetterman

> 给我看你的流程图而藏起你的表，我将仍然莫名其妙。如果给我看你的表，那么我将不再需要你的流程图，因为它们太明显了。
>
> —— Fred Brooks，《The Mythical Man-Month》（人月神话）

6.1 简介

当前大多数计算机科学的学生将Fred Brooks的这句话理解为：“给我看你的代码而藏起你的数据结构……”信息架构师坚信，处于大多数系统核心的是数据，而不是算法。随着Web的兴起，用户产生和消费的数据比以往更加推动了信息技术的使用。Web用户不会去接触QuickSort（快速排序）。他们会访问一个数据仓库。

这些数据可以是通用的，如一本电话簿；也可以是私有的，如一个在线仓库；也可以是个人的，如一个博客；也可以是开放的，如当地的天气情况；还可以是严格保护的，如在线银行记录。在任何情况下，Web呈现的几乎所有面对用户的功能归根结底都是提供一个界面，访问站点专有的一组核心数据。这些信息构成了几乎所有网站的核心价值，不论它是由顶级员工研究团队创建的还是由世界各地的用户创建的。数据推动了用户喜欢的产品，所以架构师围绕数据创建了其余的传统“*n*层”软件栈（逻辑层与显示层）。

这个故事讲的是Facebook的数据，以及它如何与Facebook平台的创建一起发展。

Facebook（*http://facebook.com*）是一个很有用的围绕数据建立架构的例子，包括用户提交的个人关系映射表、传记信息，以及文本或其他媒体内容。Facebook的工程师在构建

站点其余部分的架构时，关注的是显示和操作这些社会关系数据。这个站点的大多数业务逻辑与这些社会关系数据密切相关，诸如对各种页面的流程和访问模式，搜索的实现，查看新闻内容，以及对内容应用可见性规则。对于用户来说，这个站点的价值直接来自于他和与他有关的人对该系统所贡献的数据的价值。

“Facebook社会关系网站”在概念上是一个标准的*n*层栈，用户的请求会从Facebook的内部库中取出数据，然后通过Facebook的逻辑进行转换，最后通过Facebook的界面输出。Facebook的工程师意识到这些数据的用处超过了这些容器的限制。Facebook平台的创建显著地改变了Facebook数据访问系统的形态，它包含的愿景远远超出了*n*层栈的分离功能，目标是以应用的形式来集成外部的系统。利用居于架构中心的用户社会关系数据，该平台开发了一组Web服务（Facebook平台应用编程接口，或Facebook API）、一门查询语言（Facebook查询语言，或FQL），以及一种数据驱动的标记语言（Facebook标记语言，或FBML），目的是将应用开发者的系统与Facebook的系统结合在一起。

随着某些数据集越来越广泛地提供出来，而且用户要求跨越多个网和桌面应用来统一使用他们的数据，阅读本章的架构师可能会发现自己已经是这样一个平台的消费者，或者围绕着自己站点的数据建立了类似的平台。本章将向读者展示Facebook以一种受控的方式向外界开放数据的过程，跟随数据演进的每一步的架构选择，以及调和数据开放与渗透在社会关系系统中独特的隐私需求的过程。它包括：

- 促进这些类型的集成。
- 将数据功能从内部栈调用移到外部可见的Web服务上（Facebook API）。
- 授权访问这个Web服务，注意保持这个社会关系系统的隐私性。
- 创建一种数据查询语言，减轻这个Web服务的新客户端的负担（Facebook FQL）。
- 创建一种数据驱动的标记语言，将应用的显示集成回Facebook，同时也支持使用其他方式不能访问的数据（Facebook FBML）。

当我们将应用的架构从分离的栈进行了足够的演进之后：

- 创建一些技术来弥补Facebook体验与外部应用体验之间的差异。

对于数据平台的使用者，本章展示了我们所做的设计决定和这些决定背后的理由。用户会话、身份认证、Web服务和各种处理应用逻辑的方式等概念将不断重复出现，它们是Web上所有这些类型的平台的主题。理解它们背后的思想为数据架构提供了巨大的实践机会，而且考虑到这些平台制造者将来可能创建的功能和形式，这种理解也相当重要。

我们鼓励数据平台制造者心里想着自己的数据集，然后从Facebook开放其数据模型的方式中学习。某些设计选择和折中可能只适合Facebook，或只适合处理有隐私保护的社会

关系数据，可能不完全适用于给定的数据集。但不管怎样，在每一步我们都给出了一个实际的问题、一个数据驱动的解决方案，以及该解决方案的高层实现。对于每个新的解决方案，我们基本上会创建一个新的产品或平台，所以在任何时候我们都必须让这个新产品符合用户的预期。反过来，我们会伴随每一步的演进创造一些新技术，有时候会改变围绕应用的Web架构。

Facebook平台的开源版本可以从*http://developers.facebook.com/*获得。就像这个版本一样，本章的代码是用PHP写的。请随意查看，不过请注意，出于清晰性的考虑，这里的代码是缩写过的。

我们从这些类型的集成的动机开始，通过一个例子来讲解一个“外部的”应用逻辑和数据（一个书店）、Facebook的社会关系数据（用户信息和朋友关系），以及它们的集成。

6.1.1 某些应用核心数据

Web应用，即使是不提供也不使用任何的数据平台，基本上仍然是由它们内部的数据来驱动的。以*http://fettermansbooks.com*为例，它一是个假想的网站，提供书籍方面的信息（如果用户感兴趣，它可能也提供购买这些书的功能）。这个站点的功能可能包括可查找的库存索引、关于每件产品的基本信息，以及用户每本书作出的评论。访问这些具体的信息构成了这个应用的核心，驱动了架构的其他部分。该站点可能使用Flash和AJAX技术，支持通过移动设备来访问，并提供一个一流的用户界面。然而*http://fettermansbooks.com*存在的根本理由是让访问者能够利用某些方法得到示例6-1中这样的核心映射关系。

例6-1：书籍数据映射的例子

```
book_get_info : isbn -> {title, author, publisher, price, cover picture}
book_get_reviews: isbn -> set(review_ids)
bookuser_get_reviews: books_user_id -> set(review_ids)
review_get_info: review_id -> {isbn, books_user_id, rating, commentary}
```

所有这些最终都实现为类似简单集合的东西，能够从一个经索引的数据表中取出。这样的书籍站点如果要有存在的价值，可能还会实现其他一些不太简单的功能，如例6-2中的简单“查找”。

例6-2：简单查找映射

```
search_title_string: title_string -> set({isbn, relevance score})
```

这些功能中包含的每个键值通常都会表现为*http://fettermansbooks.com*上的一个或多个页面——有一组特有的逻辑围绕着这批数据，通过一种特有的方式显示出来。例如，要查看评论者*X*提交的一些评论，*http://fettermansbooks.com*的用户可能会被引向页面 *fettermansbooks.com/reviews. php? books_user_id=X*，或者要看ISBN号为*Y*的某本书的所有信息（包括所有对个人评论页面的链接），用户会被引向页面 http:// fettermansbooks. com/ *book.php?isbn=Y*。

像*http://fettermansbooks.com*这样的站点有一个特点是值得注意的，即几乎所有数据都对所有用户开放。它在`book_get_info`这样的映射中生成所有的内容，帮助用户发现有关某本书的尽可能多的信息。这对于一个卖书的站点可能是好事，但在接下来的使用社会关系数据的例子中，可见性限制决定了数据访问层的许多架构考虑。

6.1.2 一些Facebook核心数据

随着所谓“ Web 2.0 ”的网络技术逐渐流行，数据在系统中的核心地位就变得更明显了。Web 2.0展现的核心主题就是它们是数据驱动的，用户本身提供了绝大部分的数据。Facebook像*http://fettermansbooks.com*一样，主要由一组核心数据映射构成，它们驱动着网站的观感和功能。这些Facebook映射的极端精简集合看起来如例6-3所示。

例6-3：社会关系数据映射示例

```
user_get_friends: uid -> set(uids)
user_get_info: uid -> {name, pic, books, current_location,...}
can_see: {uid_viewer, uid_viewee, table_name, field_name} -> 0 or 1
```

这里的uid指的是（数字化的）Facebook用户标识符，从`user_get_info`返回的info指的是用户的描述信息（参见Facebook开发文档中的users.getInfo），可能包含了用户最喜欢的书籍名称，因为他们曾在*http://facebook.com*上输入过。这个系统从核心上来看与*http:// fettermansbooks.com*区别不大，只有中心数据不同，因此站点的功能也不同，这些功能围绕着用户与其他用户的联系（“ 朋友 ”），用户的内容（“ 描述信息 ”），以及内容的可视法则（“ can_see ”）。

这个`can_see`数据集是很特殊的。Facebook对于用户生成的数据有一个非常核心的隐私概念，即用户*X*查看用户*Y*的信息的业务规则。这种数据从不直接可见，但它驱动了一些重要的考虑，当我们查看外部应用的逻辑、数据与Facebook的逻辑、数据集成的例子时，会看到这些考虑反复出现。就其本身而言，Facebook到处使用这种数据集令它与*http://fettermansbooks.com*这样的站点区别开来。

Facebook平台和其他社会关系平台认识到这种社会关系映射是有用的，这种用处不仅体现在*http://facebook.com*这样的站点内部，也体现在与*http://fettermansbooks.com*这样的站点功能进行集成时。

6.1.3 Facebook的应用平台

对于*http://fettermansbooks.com*和*http://facebook.com*的共同用户来说，此时因特网应用的图景如图6-1所示。

在一般的*n*层架构中，应用将输入（对于Web来说，就是GET、POST和cookie信息的集合）映射为对原始数据的请求，这些原始数据可能存在于数据库中。它们被转换为内存中的数据，并通过一些业务逻辑进行智能化处理。输出模块将针对显示对这些数据对象进行转换，变成HTML、JavaScript、CSS等。这里，在图的顶部，是运行在基础设施之上的应用程序*n*层栈。在应用出现在Facebook平台之前，Facebook完全运行在同样的架构上。重要的是，在两个架构中，业务逻辑（包括Facebook的隐私）实际上都是根据一些规则来执行的，这些规则建立在系统的某些数据组件之上。

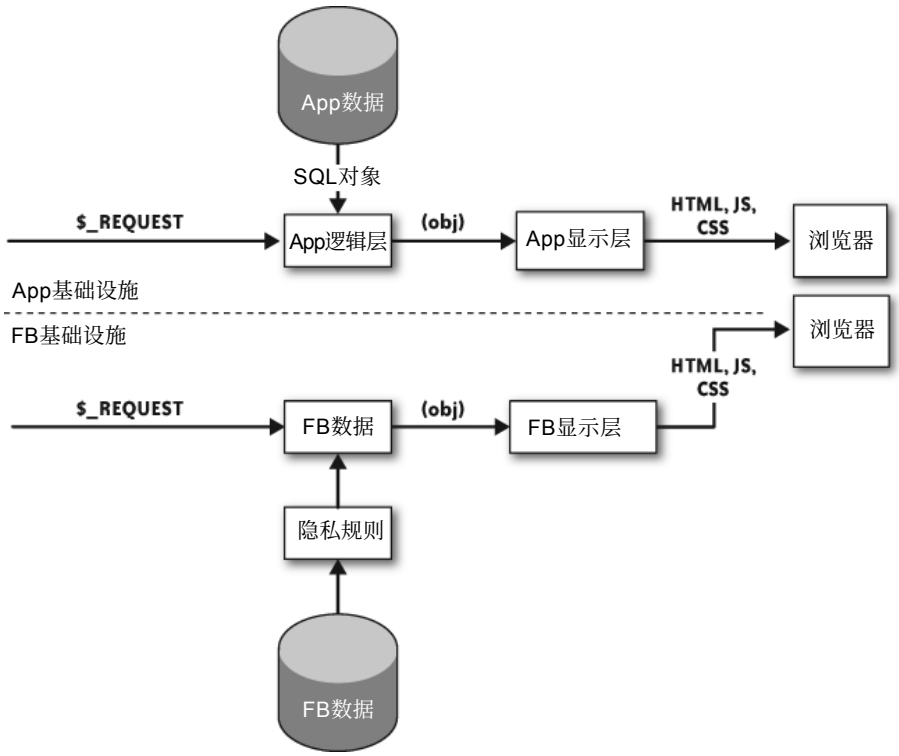

图6-1：分离的Facebook和*n*层应用栈

更大量的相关数据意味着业务逻辑可以提供更多个人定制的内容，所以在*http://fettermansbooks.com*（或其他应用）上浏览书籍，写书评、阅读或购买的体验，会被来自Facebook的用户社会关系数据加强和放大。具体来说，显示朋友的书评、期望清单和购买情况将有助于用户的购买决定，发现新的书籍，或强化与其他用户之间的联系。如果Facebook的内部映射`user_get_friends`可以由*http://fettermansbooks.com*这样的其他外部应用访问，就会为这些原本分离的应用提供强大的社会关系上下文，让应用程序不需要创建它自己的社会关系网络。所有这种类型的应用都可以与这种数据进行很好的

集成，因为开发者可以将这些核心Facebook映射应用于无数其他Web应用，用户在这些应用里提供或消费内容。

Facebook平台的技术通过在社会关系网络和数据架构方面的一系列改进，实现了这一点：

- 应用可以通过Facebook平台的数据服务来访问有用的社会关系数据，为外部的Web应用、桌面操作系统应用和其他设备上的应用提供社会关系上下文。
- 应用可以通过一种名为FBML的数据驱动标记语言来实现显示，在*http://facebook.com*的页面上集成他们的应用体验。
- 通过FBML所要求的架构改变，开发者可以使用Facebook平台的cookie和Facebook JavaScript（FBJS），从而让应用出现在*http://facebook.com*上所需的改动最小。
- 最后，应用可以获得这些功能，同时不必牺牲隐私，也不必放弃对于Facebook为用户数据和显示提供的用户体验的期望。

最后一点是最有趣的。Facebook平台的架构并非一直是美丽的——它主要被看成是社会关系平台领域的先行者。大多数的架构考虑是为了创建统一可用的社会关系上下文，它体现了这样的阴阳关系：数据可获得性和用户隐私。

6.2 创建一个社会关系Web服务

回过头来看一看像*http://fettermansbooks.com*这样一个简单的例子，我们就很清楚大多数因特网应用都会因为在数据显示时添加社会关系上下文而受益。但是，我们会遇到一个实际的问题：这种数据的可获得性。

实际问题：应用可以利用在Facebook上的用户社会关系数据，但这种数据是不可访问的。

数据解决方案：通过一个外部可以访问的Web服务来提供Facebook数据（图6-2）。

为Facebook架构添加了Facebook API，就开始通过Facebook平台为外部应用和Facebook建立了关系，本质上为外部应用栈添加了Facebook数据。对于Facebook用户，当他显式地授权外部应用可以代表他获得社会关系数据时，这种集成就开始了。

例6-4展示了*http://fettermansbooks.com*的登录页面在没有Facebook集成的情况下可能的样子。

例6-4：书籍网站逻辑示例

```
$books_user_id = establish_booksite_userid($_REQUEST);
$book_infos = user_get_likely_books($books_user_id);
display_books($book_infos);
```

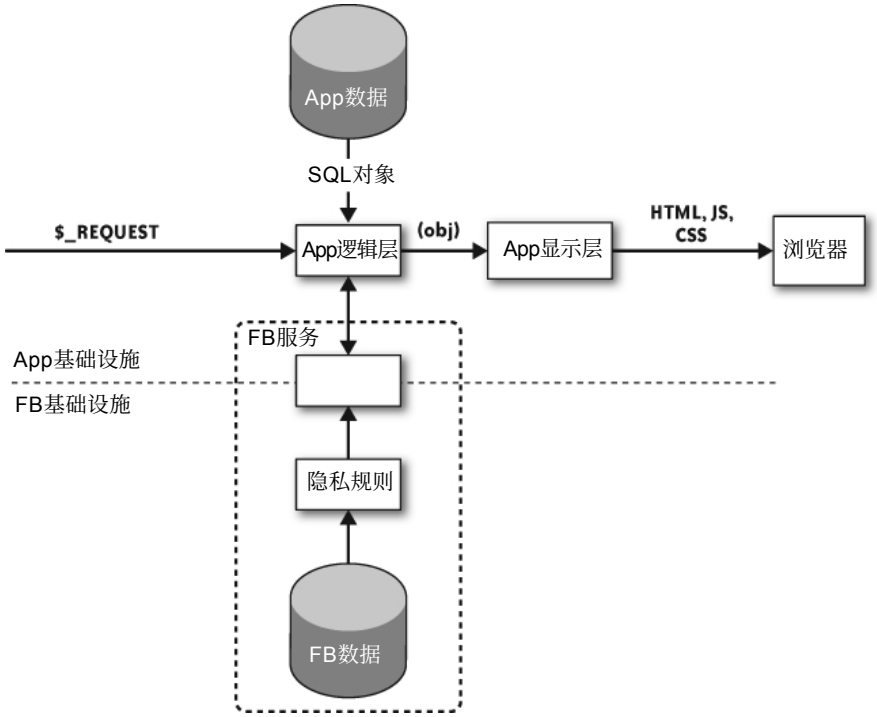

图6-2：应用栈通过Web服务使用Facebook数据

这个`user_get_likely_books`函数操作完全源自于这个书籍应用控制的数据，可能使用智能的关联技术来猜测用户的兴趣。

但是，假定Facebook为在其他站点的用户提供了两个简单的远程过程调用（RPC）方法：

- friends.get()
- users.getInfo($users, $fields)

通过它们，并添加从*http://fettermansbooks.com*的用户标识符到Facebook的用户标识符的映射关系，我们就可以为*http://fettermansbooks.com*上的所有内容添加社会关系上下文。请考虑这个针对Facebook用户的新流程，如例6-5所示。

例6-5：包含社会关系上下文的书籍站点逻辑

```
$books_user_id = establish_booksite_userid($_REQUEST);
$facebook_client = establish_facebook_session($_REQUEST,$books_user_id);

if ($facebook_client) {
  $facebook_friend_uids = $facebook_client->api_client->friends_get();
  foreach($facebook_friend_uids as $facebook_friend) {
    $book_site_friends[$facebook_friend]
```

```
        = books_user_id_from_facebook_id ($facebook_friend);
  }
  $book_site_friend_names = $facebook->api_client->
      users_getInfo($facebook_friend_uids, 'name');

  foreach($book_site_friends as $fb_id => $booksite_id) {
    $friend_books = user_get_reviewed_books($booksite_id);
     print "<hr>" . $book_site_friend_names[$fb_id] . "'s likely picks:
<br>";
    display_books($friend_books);
  }
}
```

这个例子中的粗体部分就是书籍应用使用Facebook平台提供的数据的代码。如果我们能够弄清楚函数`establish_facebook_session`背后的代码，这个架构就可以提供更多的数据，从而将这个了解书籍的应用变成了一个完全了解用户的应用。

让我们来看看Facebook的API如何支持这一点。首先，我们会简单浏览一下Web服务包装Facebook数据的技术，这是通过使用合适的元数据以及名为Thrift的灵活的代码生成器来生成的。开发者可以使用下一节中提到的这些技术，有效地创建各种Web服务，不论开发者手中的数据是公有的还是私有的。

但是请注意，Facebook的用户并不认为他们的Facebook数据全部是公有的。所以在技术概述之后，我们会探讨Facebook层面的隐私，这是通过平台API中的主要认证方式来实现的，即用户会话。

6.2.1 数据：创建一个XML Web服务

为了能够在一个示例应用中提供基本的社会关系上下文，我们已经建立了两个远程方法调用，即`friends.get`和`users.getInfo`。访问这些数据的内部功能可能存在于Facebook代码树的某个库中，为Facebook站点上的类似请求提供服务。例6-6展示了一些例子。

例6-6：社会关系映射示例

```
function friends_get($session_user) { ... }
function users_getInfo($session_user, $input_users, $input_fields) { ... }
```

我们接下来要创建一个简单的Web服务，将通过HTTP的GET和POST输入转换成对内部栈的调用，以XML的格式输出结果。在Facebook平台中，目标方法的名称以及它的参数是在HTTP请求中传递的，还包括一些与调用应用相关的证书（称为“api key”），与用户-应用对相关的证书（称为“用户会话key”），与请求实例本身相关的证书（称为请求“签名”）。我们稍后将在6.2.2节中讨论会话key。要服务一个针对http://api.facebook.com的请求，其大致过程如下：

1. 检查传递的证书（第6.2.2节），验证调用应用程序的身份，用户当前在该应用中

的授权，以及请求的可信度。

2. 将进入的GET/POST请求解释为带有相应参数的方法调用。

3. 对内部方法进行单次调用，将结果保存为内存中的数据结构。

4. 将这些数据结构转换成已知的输出格式（如XML或JSON）并返回。

创建外部可使用的接口，难度主要在于第2步和第4步。为外部使用者提供这些数据接口的一致维护、同步和文档是很重要的，手工打造一个代码框架来确保这种一致性则是一项无人赞赏而又耗时的工作。另外，我们可能需要将这些数据提供给多种语言编写的内部服务来使用，或者以不同的Web协议将结果提供给外部开发者，如XML、JSON或SOAP。

那么这里的优美解决方案，就是利用元数据来封装数据类型和描述API的方法签名。Facebook的工程师创建开源的跨语言进程间通信（IPC）系统，名为Thrift（*http://developers.facebook.com/thrift*），干净利落地实现了这个目标。

深入一步，例6-7展示了一个针对1.0版API的“.thrift”文件的例子，在这个版本里，Thrift包实现了这个API的大部分机制。

例6-7：通过Thrift对Web服务定义

```
xsd_namespace http://api.facebook.com/1.0/
/***
* Definition of types available in api.facebook.com version 1.0
*/
typedef i32 uid
typedef string uid_list
typedef string field_list

struct location {
 1: string street xsd_optional,
 2: string city,
 3: string state,
 4: string country,
 5: string zip xsd_optional
}

struct user {
 1: uid uid,
 2: string name,
 3: string books,
 4: string pics,
 5: location current_location
}

service FacebookApi10 {

 list<uid> friends_get()
  throws (1:FacebookApiException error_response),
```

```
 list<user> users_getInfo(1:uid_list uids, 2:field_list fields)
  throws (1:FacebookApiException error_response),
}
```

这个例子中的类型是原生类型（string）、结构（location、user）或泛型方式的集合（list<uid>）。因为每个方法描述都有精心设计类型的方法签名，定义复用的类型的代码就可以直接在任何语言中生成。例6-8展示了针对PHP的部分生成结果。

例6-8：Thrift生成的服务代码

```
class api10_user {

public $uid = null;
public $name = null;
public $books = null;
public $pic = null;
public $current_location = null;


public function __construct($vals=null) {
  if (is_array($vals)) {
    if (isset($vals['uid'])) {
       $this->uid = $vals['uid'];
     }
     if (isset($vals['name'])) {
       $this->name = $vals['name'];
     }
     if (isset($vals['books'])) {
       $this->books = $vals['books'];
     }
     if (isset($vals['pic'])) {
       $this->pic = $vals['pic'];
     }
     if (isset($vals['current_location'])) {
       $this->current_location = $vals['current_location'];
    }
   // ...
  }
 // ...
}
```

返回user类型的所有内部方法都会创建全部需要的字段，结束的语句类似例6-9的样子。

例6-9：一致地使用生成的类型

```
return new api_10_user($field_vals);
```

例如，如果current_location（当前位置）出现在这个用户对象中，那么$field_vals['current_location']就会在例6-9的代码执行之前，被赋值为new api_10_user(...)。

字段的名称和类型本身实际上会生成XML输出所需的schema，以及相应的XML Schema文档（XSD）。例6-10展示了整过RPC过程实际输出的XML。

例6-10：Web服务调用的XML输出

```
<users_getInfo_response list="true">
 <users type="list">
  <user>
   <name>Dave Fetterman</name>
   <books>Zen and the Art, The Brothers K, Roald Dahl</books>
   <pic></pic>
   <current_location>
    <city>San Francisco</city>
    <state>CA</state>
    <zip>94110</zip>
   </current_location>
  </user>
 </users>
</users_getInfo_response>
```

Thrift生成类似的代码来声明RPC函数调用、序列化成已知的输出格式，并将内部的异常转化成外部错误代码。其他像XML-RPC或SOAP这样的工具集也提供这样一些好处，但可能需要更多的CPU和带宽开销。

使用像Thrift这样的漂亮工具有以下好处：

自动化类型同步

在user类型中添加“`favorite_records`”，或将`uid`转换成i64需要在所有使用或生成这些类型的方法中进行。

自动化绑定生成

所有读写类型的麻烦工作都不需要了，转换函数调用生成XML的RPC方法要求函数声明、类型检查和错误处理，这些都由Thrift自动完成。

自动化文档

Thrift生成公开的XML Schema文档，它将作为外界看到的无二义的文档，通常比在“手册”上看到的文档要好得多。这种文档也可以直接在一些外部工具中使用，生成客户端的绑定。

跨语言同步

这个服务可以由外部的XML客户端或JSON客户端调用，内部是通过各种语言（PHP、Java、C++、Python、Ruby、C#等）写的服务程序通过套接口来通信的。这要求基于元数据的代码生成，这样服务的设计者就不必在每次小改动时花时间更新这些代码。

我们已经有了社会关系网站服务的数据组件。接下来我们将弄清楚如何建立这些会话键，在所有Facebook扩展上强制实现用户期望的隐私模型。

6.2.2 简单的Web服务认证握手

一个简单的认证策略让我们能够在尊重Facebook用户的隐私观点的前提下访问这些数据。用户对Facebook系统的数据有某种特定的视图，这取决于用户是谁、用户的隐私设定，以及与用户有关系的人的隐私设定。用户可以授权单个应用来继承这一视图。用户通过某个应用可以看到的信息，是用户通过Facebook可以看到的信息中有意义的一部分（但不会超出通过Facebook可以看到的信息）。

在独立应用站点的架构中（图6-1），用户认证通常采用浏览器发送cookie的方式，这些cookie是该站点在最初执行过认证动作之后生成的。但是在图6-2中，通常作为Facebook用法一部分的`cookie`不再提供了——外部应用需要在没有用户浏览器的帮助下从Facebook平台请求信息。为了修正这一点，我们在会话键映射的基础上设计Facebook，如例6-11所示。

例6-11：会话键映射

```
get_session: {user_id, application_id} -> session_key
```

Web服务的客户端只要在每次请求时发送`session_key`，让Web服务知道这代表的是哪个用户的请求执行。如果用户（或Facebook）禁用了这个应用，或者他从未用过这个应用，安全检查就会通不过，会返回一个错误。否则，外部应用站点会把这个会话键记入它自己的用户记录，或者放到该用户的cookie中。

但在最开始如何得到这个会话键呢？在*http://fettermansbooks.com*应用代码中的`establish_facebook_session`是一个占位符，为这个过程保留的。每个应用都有它自己特有的“应用键”（也称为`api_key`），开始应用认证流程（图6-3）：

1. 用户通过一个已知的`api_key`重定向到Facebook登录界面。
2. 用户在Facebook上输入口令，对这个应用授权。
3. 用户带着会话键和用户ID重定向到已知的应用。
4. 应用现在获得了授权，可以代表用户调用API方法（除非会话超时或被删除）。

要帮助用户发起这个流程，可以使用下面包含应用键（即“`abc123`”）的链接或按钮：

```
<a href="http://www.facebook.com/login.php?api_key=abc123">
```

如果用户通过Facebook上口令输入同意授权给这个应用（注意，口令是Facebook最需要保护的数据），用户就被重定向回这个应用站点，带着有效的会话键和Facebook用户ID。这个会话键是非常私密的，所以对于将来的验证，应用的所有调用都会带有从这个共享秘密生成的散列值。

假定开发者隐藏了他的`api_key`和应用私密数据，`establish_facebook_session`可

以很简单地按图6-3中的流程来编写。尽管这种类型的系统握手的细节可以不同，但重要的是只有当用户在Facebook上的关键步骤中输入了他的口令，才会产生授权。很有趣的是，一些早期的应用只是使用了这种认证握手来作为它们的口令系统，而根本没有使用其他的Facebook数据。

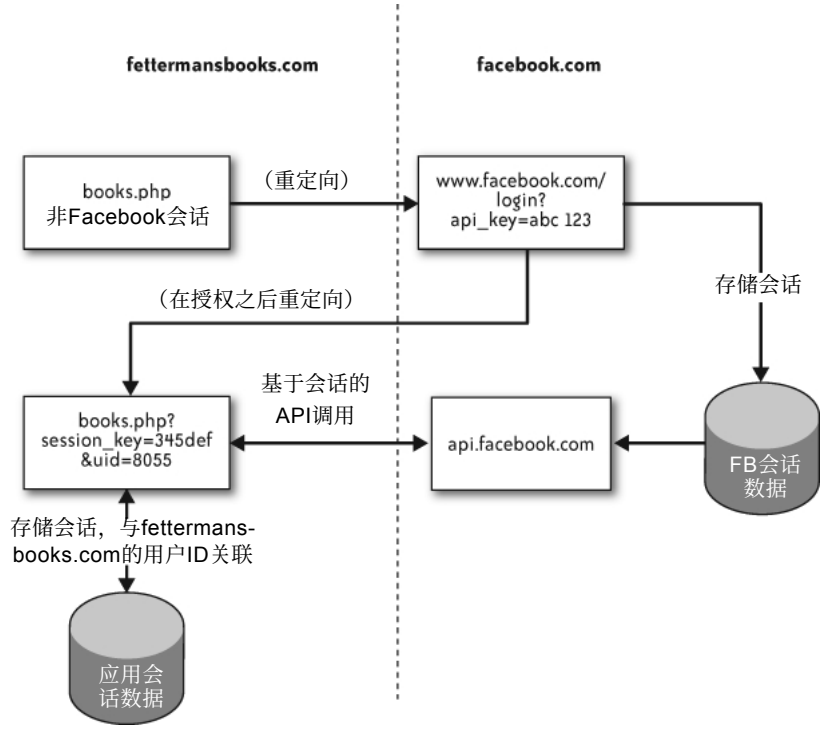

图6-3：对Facebook平台API的认证访问

但是，某些应用不容易适应这种第二步“重定向”的方式。“桌面”风格的应用、基于设备的应用（如手机应用），或浏览器内建的应用有时候也相当有用。在这种情况下，我们采用一种稍微不同的机制来使用第二次认证令牌。令牌是应用通过API请求得到的，在第一次登录时传递给Facebook，然后在现场用户认证之后，应用换到一个会话键和会话专有的一些私密信息。

6.3 创建社会关系数据查询服务

通过一个带有用户控制的认证握手的Web服务，我们已经将我们的内部库扩展到外部世界。通过这个简单的改变，Facebook的社会关系数据现在可以驱动其用户决定认证的任何其他应用程序，通过普遍关注的社会关系上下文，在应用的数据中创建新的关系。

随着用户渐渐了解这些数据交换的无缝性，使用这些平台API的开发者知道这些数据集是很独特的。开发者访问自己的数据的模式与访问Facebook数据的模式有着很大的不同。例如，Facebook的数据位于HTTP请求的另一端，通过许多HTTP连接来调用这些方法增加了开发者自己页面的延迟和开销。他自己的数据库也提供了更大粒度的访问，优于Facebook平台API中的几十个方法。使用他自己的数据和SQL这样熟悉的查询语言，他可以选择一个表的特定字段，对结果集排序或进行限制，匹配其他的指标，或进行嵌套查询。如果平台的API不能够让开发者在平台的服务器上进行智能的处理，开发者就必须经常获取相关数据的超集，收到数据后再在他自己的服务器上进行这些标准的逻辑转换。这可能成为严重的负担。

实际问题：从Facebook平台API获取数据要比获取内部数据的开销大很多。

随着应用越来越多地使用外部数据平台，诸如带宽占用、CPU负载和请求延迟等因素很快累积起来。难道我们没有在自己的单个应用栈的数据层中对此进行优化吗？没有技术让我们通过一次调用取得多个数据集吗？如果在这个数据层中进行选择、限制和排序，结果会怎样？

数据解决方案：类似内部数据采用的模式，实现外部数据访问模式：一种查询服务。

Facebook的解决方案称为FQL，我们将在6.3.2节中详细介绍。FQL很像SQL，但它将平台数据转换成字段和表，而不是简单松散地定义为XML schema中的对象。这让开发者能够在Facebook的数据上使用标准的数据查询语义，这种方式可能与他们取得自己数据的方式一样。同时，将计算推到平台一端的好处与将操作通过SQL推到数据层的好处是相似的。在这两种情况下，开发者有意识地避免了在应用逻辑中进行这种处理的代价。

FQL代表了基于Facebook的内部数据的另一项数据架构改进，是标准的黑盒Web服务的进步。但是首先，我们先来看一种容易而明显的方法，它让开发者能够消除多次数据请求的来回开销，同时我们也要说明为什么这是不够的。

6.3.1 批量方法调用

对于负载问题最简单的解决方案，就是类似于Facebook的`batch.run`API方法。这消除了多次通过HTTP栈对*http://api.facebook.com*进行调用的来回开销，一批接受多个方法调用的输入，一次返回输出的多棵XML树。在客户端，这个过程转变成类似例6-12中的代码。

例6-12：批量方法调用

```
$facebook->api_client->begin_batch();
$friends = &$facebook->api_client->friends_get();
$notifications = &$facebook->api_client->notifications_get();
$facebook->api_client->end_batch();
```

在Facebook平台的PHP5客户端库中，end_batch实际上是向平台服务器发起请求，取得所有结果，并针对每个结果更新引用的变量。这里我们从一次用户会话中批量获取了用户数据。通常，人们用批量查询机制将许多设置操作归为一组，如大量的Facebook个人描述更新，或大量突发的用户通知。

这些批量操作很有效，但这也揭示了这种批量操作的主要问题。问题是，每次调用必须与其他调用的结果无关。对多个不同用户的批量操作通常具备这种特点，但有一种常见的情况仍然不能处理，即使用一次调用的结果作为下次调用的输入。例6-13展示了不能利用批量机制的一种常见情况。

例6-13：批量机制的不正确用法

```
$fields = array('uid, 'name', 'books', 'pic', 'current_location');
$facebook->api_client->begin_batch();
$friends = &$facebook->api_client->friends_get();
$user_info = &$facebook->api_client->users_getInfo($friends, $fields); // NO!
$facebook->api_client->end_batch();
```

当客户端发出users_getInfo请求时，$friends的内容显然还不存在。FQL模型优雅地解决了这个问题和其他问题。

6.3.2 FQL

FQL是一种简单的查询语言，它包装了Facebook的内部数据。输出的格式通常与Facebook平台API的输出格式一样，但输入超出了简单的RPC库的模型，变成了SQL的查询模型：命名的表和字段，包含已知的关系。像SQL一样，这种技术添加了选择实例或范围的能力，从数据行中选择字段子集的能力，并通过嵌套查询将更多的工作推到数据服务器端，避免了通过RPC栈进行多次调用。

举个例子，如果期望的输出是所有用户中我朋友的“uid”、“name”、“book”、“pic”和“current_location”字段，在我们的纯API模型中，我们会使用例6-14中的过程。

例6-14：在客户端串联方法调用

```
$fields = array('uid', 'name', 'books', 'pic', 'current_location');
$friend_uids = $facebook->api_client->friends_get();
$user_infos = users_getInfo($friend_uids, $fields);
```

这导致了对数据服务器的多次调用（这里是2次），更大的延迟，更大的失败可能性。相反，对于查看用户编号8055（实际上是你的），我们在例6-15中写出这样的FQL语法并进行一次调用。

例6-15：利用FQL在服务器端串联方法调用

```
$fql = "SELECT uid, name, books, pic, current_location FROM profile
        WHERE uid IN (SELECT uid2 from friends where uid1 = 8055)";
$user_infos = $facebook->api_client->fql_query($fql);
```

我们在概念上将users_getInfo引用到的数据视为一个表，它基于一个索引（uid），包含一些可选择的字段。如果正确地扩展，这种新的语法可以支持一些新的数据访问能力：

- 限定范围查询（例如根据事件发生的时间）。
- 嵌套查询（SELECT fields_1 FROM table WHERE field IN (SELECT fields_2 FROM ....)）。
- 结果集大小限制和排序 。

FQL的架构

开发者通过fql_query API来调用FQL。问题的要点是在FQL的命名“ 表 ”和“ 字段 ”中，统一外部API的命名“ 对象 ”和“ 属性 ”。我们仍然继承了标准API的流程：通过内部方法取得数据，应用跟这个方法的API调用相关的规则，然后根据第6.2.1节介绍的Thrift系统，转换到输出。对于每个数据读取API方法，在FQL中都有一个对应的“ 表 ”，代表了这次查询背后的数据抽象。例如，API方法users_getInfo，它提供给定用户ID的姓名、照片、书籍和当前位置等字段，在FQL中它就表现为用户表和对应的字段。fql_query的输出实际上也符合标准API的输出（如果修改XSD来允许省略对象小的字段），所以在用户表上调用fql_query返回的输出与相应的users_getInfo调用是等价的。事实上，像user_getInfo这样的调用在Facebook的服务器端通常是实现为FQL调用的！

注意：在编写本章时，FQL只支持SELECT，不支持INSERT、UPDATE、REPLACE、DELETE和其他操作，所以只有读取方法可以通过FQL来实现。大多数操作这类数据的Facebook平台API方法现在是只读的。

我们从这个用户表开始，以它为例，创建FQL系统来支持对它的查询。在平台的各个数据抽象层之下（内部调用、users_getInfo外部API调用，以及新的FQL的用户表），想象Facebook在自己的数据库中有一个名为“ user ”的表（例6-16）。

例6-16：Facebook数据表示例

```
> describe user;
+--------------+--------------+-----+
| Field        | Type         | Key |
+--------------+--------------+-----+
| uid          | bigint(20)   | PRI |
| name         | varchar(255) |     |
| pic          | varchar(255) |     |
| books        | varchar(255) |     |
| loc_city     | varchar(255) |     |
| loc_state    | varchar(255) |     |
| loc_country  | varchar(255) |     |
| loc_zip      | int(5)       |     |
+--------------+--------------+-----+
```

在Facebook的程序栈中，支持我们访问这个表的方法是：

```
function user_get_info($uid)
```

它在我们选择的语言（PHP）中返回一个对象，通常此后再应用隐私逻辑，并展现在*http://facebook.com*上。我们的Web服务实现做的事情相当类似，将Web请求的GET/POST内容转给这样一个调用，得到类似的栈对象，应用隐私逻辑，然后通过Thrift将它变成一个XML响应（图6-2）。

我们可以在FQL中将`user_get_info`包装起来，实际实现这个模型，将表、字段、内部函数和隐私组织成一个逻辑上的、可重复的形式。

下面是例6-15中的FQL调用创建的一些关键对象，以及描述它们的关系的方法。讨论所有的字符串解析、语法实现、可选索引、交集查询和实现许多不同的组合表达式（比较、“in”语句、交集、非交集）超出了本章的范围。这里我们只是关注面向数据的部分：FQL中数据的对应字段和表对象的高层规范，并将查询输入语句转换每个字段的`can_see`和`evaluate`函数（例6-17）。

例6-17：FQL字段和表示例

```
class FQLField {
  // e.g. table="user", name="current_location"
  public function __construct($user, $app_id, $table, $name) { ... }

  // mapping: "index" id -> {0,1} (visible or invisible)
  public function can_see($id) { ... }

  // mapping: "index" id -> Thrift-compatible data object
  public function evaluate($id) { ... }
}

class FQLTable {
  // a static list of contained fields:
  // mapping: () -> ('books' => 'FQLUserBooks', 'pic' ->'FQLUserPic', ...)
  public function get_fields() { ... }
}
```

`FQLField`和`FQLTable`对象构成了这个访问数据的新方法。`FQLField`包含了针对数据的逻辑，将“行”（如用户ID）和查看者的信息（用户和`app_id`）转换成我们内部的栈数据调用。在此之上，我们确保隐私评估利用要求的`can_see`方法得以正确实现。在我们处理一个请求时，我们可以在内存中为每个命名的表格（“`user`”）创建这样一个`FQLTable`对象，为每个命名的字段创建一个`FQLField`对象（为“`books`”创建一个，为“`pic`”创建一个，等等）。对应到一个`FQLTable`中的每个`FQLField`对象一般会使用底层相同的数据访问程序（在下面的例子里，是`user_get_info`），虽然不一定是这样——这只是一个方便的接口。例6-18展示了用户表中典型的字符串字段的例子。

例6-18：将核心数据库映射到FQL字段定义

```
// base object for any simple FQL field in the user table.
class FQLStringUserField extends FQLField {

  public function __construct($user, $app_id, $table, $name) { ... }

  public function evaluate($id) {
    // call into internal function
    $info = user_get_info($id);
    if ($info && isset($info[$this->name])) {
      return $info[$this->name];
    }
    return null;
  }

  public function can_see($id) {
    // call into internal function
    return can_see($id, $user, $table, $name);
  }
}

// simple string data field
class FQLUserBooks extends FQLStringUserField { }

// simple string data field
class FQLUserPic extends FQLStringUserField { }
```

FQLUserPic和FQLUserBooks的区别仅限于它们的内部属性$this->name，这是由它们的构造方法在处理过程中设置的。请注意，在底层，我们针对表达式中需要的每次求值调用user_get_info；只有系统将这些结果缓存在内存中，才能取得较好的性能。Facebook的实现就是这样做的，整个查询执行的时间与标准平台API调用的时间是同一量级的。

下面是一个更复杂的字段，表示current_location，它采用的是同样的输入，展示了同样的使用模式，但输出了一个我们前面曾看到过的结构类型对象（例6-19）。

例6-19：更复杂的FQL字段映射

```
// complex object data field
class FQLUserCurrentLocation extends FQLStringUserField {
  public function evaluate($id) {
    $info = user_get_info($id);
    if ($info && isset($info['current_location'])) {
      $location = new api10_location($info['current_location']);
    } else {
      $location = new api10_location();
    }
    return $location;
  }
}
```

像api10_location这样的对象是6.2.1小节中所说的生成的类型，Thrift和Facebook数据服务知道如何将它返回为良好类型的XML。现在我们知道，为什么就算是新的输入形式，FQL的输出也不会与Facebook API产生不兼容的情况。

在下面的例子中，FQLStatement的主要求值循环告诉了我们FQL实现的大致思想。在这段代码中我们引用了FQLExpression，但在简单的查询中，我们更有可能提到的是FQLFieldExpression，它包装了对FQLField自己的求值和can_see方法的内部调用，如例6-20所示。

例6-20：一个简单的FQL表达式类

```
class FQLFieldExpression {

  // instantiated with an FQLField in the "field" property
  public function evaluate($id) {
    if ($this->field->can_see($id))
      return $this->field->evaluate($id);
    else
      return new FQLCantSee(); // becomes an error message or omitted field
  }

  public function get_name() {
    return $this->field_name;
  }
}
```

要发起整个流程，类似SQL的字符串输入通过lex和yacc转换成主要FQLStatement的$select表达式数组和$where表达式。FQLStatemenet的evaluate()函数将返回我们请求的对象。例6-21中的主语句求值循环包括了以下步骤，说明了简单的大致顺序：

1. 取得我们希望返回的行在索引上的约束。例如，如果在用户表上选取，这就是我们想查询的那些UID。如果我们在一个按时间索引的事件表上查询，这就是时间边界。
2. 将这些转换成表的规范ID。用户表也可以按字段名查询，如果FQL表达式使用了字段名称，这个函数就会使用内部的user_name到user_id的查找函数。
3. 针对每个候选ID，看看它是否满足RHS表达式子句（布尔逻辑、比较、“ IN ”操作等）。如果不满足，就抛弃它。
4. 对每个表达式求值（在我们的例子里，就是SELECT子句中的字段），然后创建<COL_NAME>COL_VALUE</COL_NAME>格式的XML元素，其中COL_NAME是FQLTable中的字段名称，COL_VALUE是字段通过它对应的FQLField的求值函数进行求值的结果。

例6-21：FQL的主求值流程

```
class FQLStatement {

  // contains the following members:
```

```
// $select: array of FQLExpressions from the SELECT clause of the query
//  corresponding to, say, "books", "pic", and "name"
// $from: FQLTable object for the source table
// $where: FQLExpression containing the constraints for the query.
// $user, $app_id: calling user and app_id

public function __construct($select, $from, $where, $user, $app_id) {... }

// A listing of all known tables in the FQL system.
public static $tables = array(
  'user'        => 'FQLUserTable',
  'friend'      => 'FQLFriendTable',
);

// returns XML elements to be translated to service output
public function evaluate() {

  // based on the WHERE clause, we first get a set of query expressions that
  // represent the constraints on values for the indexable columns contained
  // in the WHERE clause

  // Get all "right hand side" (RHS) constants matching ids (e.g. X, in 'uid = X')
  $queries = $this->where->get_queries();

  // Match to the row's index. If we were using 'name' as an alternative index
  // to the user table, we would transform it here to the uid.
  $index_ids = $this->from_table->get_ids_for_queries($queries);

  // filter the set of ids by the WHERE clause and LIMIT params
  $result_ids = array();

  foreach ($ids as $id) {
    $where_result = $this->where->evaluate($id);

      // see if this row passes the 'WHERE' constraints
      // is not restricted by privacy
      if ($where_result && !($where_result instanceof FQLCantSee))
        $result_ids []= $id;
  }

  $result = array();
  $row_name = $this->from_table->get_name(); // e.g. "user"

  // fill in the result array with the requested data
  foreach ($result_ids as $id) {
    foreach ($this->select as $str => $expression) { // e.g. "books" or "pic"
      $name = $expression->get_name();
      $col = $expression->evaluate($id); // returns the value
      if ($col instanceof FQLCantSee)
        $col = null;

      $row->value[] = new xml_element($name, $col);
    }
```

```
      $result[] = $row;
    }
    return $result;
}
```

FQL还有其他一些精妙之处，但这个总体流程说明了已有的内部数据访问和隐私规则实现与全新的查询模型的结合。这让开发者能够更快地处理它的请求，能够以比API更好的粒度来访问数据，同时又保持了SQL类似的语法。

由于我们的许多API在内部包装了对应的FQL方法，我们的整体架构演变为图6-4所示的状况。

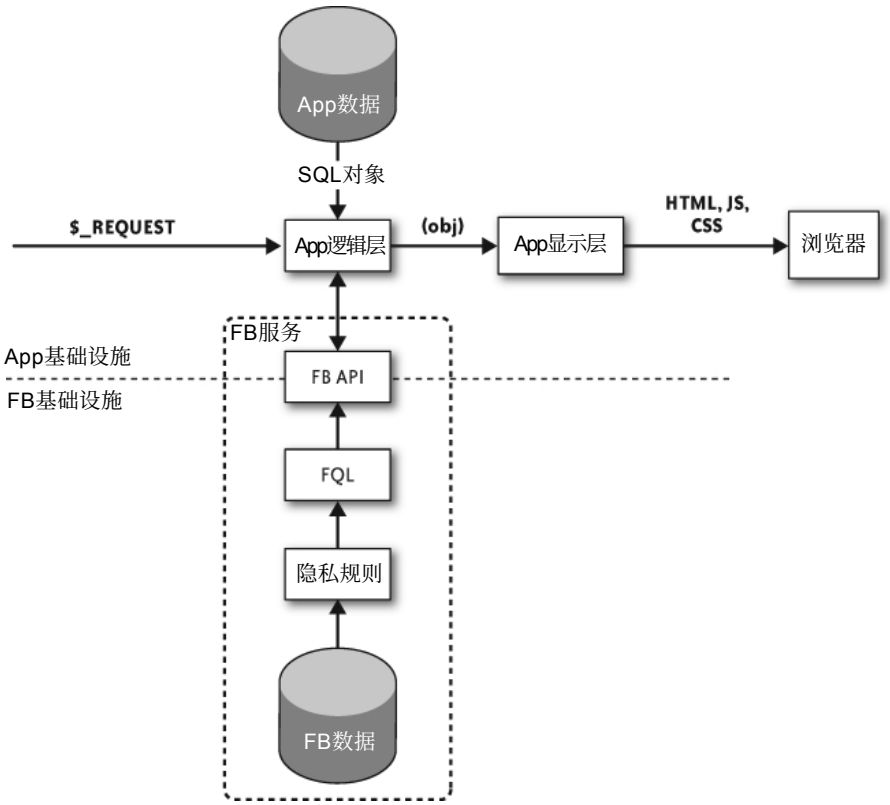

图6-4：通过Web和查询服务来使用Facebook数据的应用栈

6.4 创建一个社会关系Web门户：FBML

前面讨论的服务让外部的应用栈能够在它们的系统中包含社会关系平台的数据，这是很大的进步。这些数据架构实现了让社会关系平台数据更开放的承诺：外部应用（如*http://fettermansbooks.com*）和数据平台（如*http://facebook.com*）的共同用户可以共享

信息，每个新的社会关系应用就不需要一个新的社会关系网络。但是，即使有了这些新的能力，这些应用还是不能享受Facebook这样的社会关系网站的全部强大功能。应用还需要让许多用户发现，才会变得有价值。而且，并不是所有支持社会关系平台的内部数据都可以提供给这些外部的应用栈。平台的创建者需要解决这些问题，我们将依次讨论。

实际问题：对于社会关系应用来说，要获得引人注目的关键性用户数，支持它的社会关系网络上的用户必须要能注意到其他用户在利用这些应用进行交互。这意味着应用与社会关系网站更深层次上的集成。

这个问题在早期的软件中就存在了：我们难以让数据、产品或系统得到广泛使用。缺少用户成为Web 2.0空间中特别值得一提的困难，因为如果没用户使用并且（特别是）生成内容，我们的系统什么时候才有用呢？

Facebook支持大量的用户，他们对在社会联系之间共享信息感兴趣，而且Facebook的特点就是把应用的内容和它自己的内容等同视之。让外部的应用出现在Facebook站点上，就会让大小开发者开发的应用更容易发现，帮助他们获得支持好的社会关系功能所需的关键性用户数。

不管我们创建什么解决方案，应用都需要在Facebook站点上有独特的显示展现。Facebook平台向应用提供了这方面的支持，为Facebook上的应用内容展现保留了URL路径*http://apps.facebook.com/fettermansbooks/…*。我们稍后会看到平台是如何集成应用的数据、逻辑和显示的。

6.2.1节和6.3.2节中介绍的数据服务衍生出第二个问题，它也同样不好处理。

实际问题：外部应用不能够使用Facebook没有通过Web服务暴露出来的那些核心数据元素。

在Facebook提供网站（*http://facebook.com*）的内容时，Facebook为它的用户提供了大量的数据。隐私信息本身（第6.1.2节中提到的`can_see`映射）就是一个好例子——不能被Facebook站点的用户显式地看到，`can_see`映射对于数据服务也是不可见的。但是强制实现Facebook用户的这种隐私设置是所有良好集成的应用的特点，也是对社会关系系统上用户期望的支持。Facebook为了保护用户的隐私，不能通过数据服务将这些数据开放出来。开发者怎样才能利用这些数据呢？

对这些问题的最优雅解决方案就是结合Facebook的数据和外部应用的数据、逻辑和显示，同时让用户在一个受信任的环境下操作。

数据解决方案：开发者通过一种数据驱动的标记语言，在社会关系站点上创建应用执行和显示的内容，与Facebook交互。

只使用第6.2.1节和第6.3.2节中介绍的Facebook平台元素的应用，在Facebook之外创造了

一种社会关系体验，因Facebook的社会关系数据而变得更强大。利用本节介绍的数据和Web架构，应用本身也变成一种数据服务，支持针对Facebook的内容显示在*http:// apps.facebook.com*之下。像*http://apps.facebook.com/fettermansbooks/...* 这样的URL不再映射到Facebook生成的数据、逻辑和显示，而是会查询*http://fettermansbooks.com*的服务，生成应用的内容。

我们必须同时记得我们的资产和约束。一方面，我们有一个访问频率很高的社会关系系统，让用户能发现外部的内容，并有大量的社会关系数据来增强这种社会关系应用。另一方面，请求需要从社会关系站点（Facebook）上发起，将应用作为服务来使用，然后将内容渲染成HTML、JavaScript和CSS，并且不违反Facebook用户的隐私或期望。

首先，我们来看一些不正确的尝试。

6.4.1 Facebook上的应用：直接渲染HTML、CSS和JS

假定一个外部应用的配置现在包含两个字段，名为`application_name`和`callback_url`。通过输入“ fettermansbooks"这样的名字和*http://fettermansbooks.com/fbapp/*这样的URL，*http://fettermanbooks.com*声明它将在自己的服务器上为用户提供服务，对*http://apps.facebook.com/fettermansbooks/PATH?QUERY_STRING*的请求将转向*http://fettermansbooks.com/fbapp/PATH?QUERY_STRING*。

对http://apps.facebook.com/fettermansbooks/…的请求于是简单地取出应用服务器上的HTML、JS和CSS等内容，并在Facebook上的页面主内容区域进行显示。这基本上是将外部站点作为一个HTML Web服务来渲染的。

这对应用的*n*层模型进行了重要改变。以前，应用栈会通过数据服务来使用Facebook的内容，这个数据服务是直接服务于对*http://fettermansbooks.com*的请求的。现在，应用在它的Web根下维护了一个树型结构，它自己提供HTML服务。Facebook通过在线请求这个新应用服务（该服务又可能用到Facebook的数据服务）而取得内容，将它包装成一般的Facebook站点导航元素，显示给用户。

但是，如果Facebook直接在它的页面中渲染一个应用的HTML、JavaScript或CSS，这就会允许应用完全违反用户对*http://facebook.com*上受控体验的期望，让站点和用户暴露在各种安全攻击之下。允许外部用户直接订制标记语言和脚本几乎从来都不是好主意。实际上，代码或脚本注入通常是攻击者的目标，所以这并不是一个很好的特征。

而且，没有新数据！尽管这为应用栈的改变奠定了基础，但这个解决方案没有完全解决前面的两个实际问题。

6.4.2 Facebook上的应用：iframe

还有一种更安全的显示应用内容的方法，可以显示另一个站点的可视化上下文和界面流转，这种方法已包含在浏览器中，即iframe。

为了复用前一节中提到的映射，对 *http://apps.facebook. com/ fettermans books/ PATH?QUERY_STRING*的请求将导致输出这样的HTML：

```
<iframe src="http://fettermansbooks.com/fbapp/PATH?GET_STRING"></iframe>
```

这个URL的内容将显示在Facebook页面的一个帧中，在它自己的沙盒环境中可以包含任何类型的Web技术：HTML、JS、AJAX、Flash等。

这实际上是让浏览器成为请求代理者，而不是由Facebook作为请求代理者。这比前一节中的模型有改进，浏览器也维护所得页面中其他元素的安全性，所以开发者可以在这个帧中随意创建他们想要的用户体验。

对于某些应用，如果开发者希望花最小的代价将他们的代码从他们的站点移到平台上，那么iframe的方式也是有意义的。实际上，Facebook继续支持完整页面生成的iframe模型。虽然这解决了第一个实际问题，将应用纳入到社会关系站点，但第二个实际问题仍未解决。虽然基于iframe的请求流程可以确保安全，但除了API服务暴露出来的数据之外，这些开发者并不能利用其他的新数据。

6.4.3 Facebook上的应用：FBML是数据驱动的执行标记语言

前两节中提到的解决方案尝试都有其优点。HTML的解决方案采用了直观的方法，将应用本身变成Web服务，将触点带回到Facebook上显示。iframe方式的好处在于将开发者的应用内容放在一个独立的（安全的）执行沙盒中。最佳解决方案将保留“应用即服务”的模型和iframe的安全和可信，同时又让开发者能够使用更多的社会关系数据。

问题是，为了让社会关系应用提供独特的使用体验，开发者必须通过他们自己的应用栈来提供数据、逻辑和展现。但是，生成这些输出必须用到那些不能离开Facebook的用户数据。

解决方案是什么？不是发回HTML，而是一种特定的标记语言，其中定义了足够的标记来表现应用的逻辑和显示，也包含对受保护数据的请求，完全让Facebook在受信任的服务器环境中渲染它！这就是FBML的前提 （图6-5）。

在这个流程中，对*http://apps.facebook.com*的请求同样被转换成对应用的请求，应用栈会使用Facebook的数据服务。但是，开发者不会让应用返回HTML，而是重写应用，返回FBML。FBML中包含了许多HTML元素，而且添加了Facebook特别定义的标签。当这个请求返回其内容时，Facebook的FBML解释器将这段标记语言转换成它自己的数据、

执行和显示实例，生成应用页面。用户就会收到一个页面，其中包含了Facebook页面的一般Web元素，而且也包含了应用的数据、逻辑和观感。不论FBML返回什么，它都能在技术上确保Facebook强制实现其隐私理念和良好的用户体验元素。

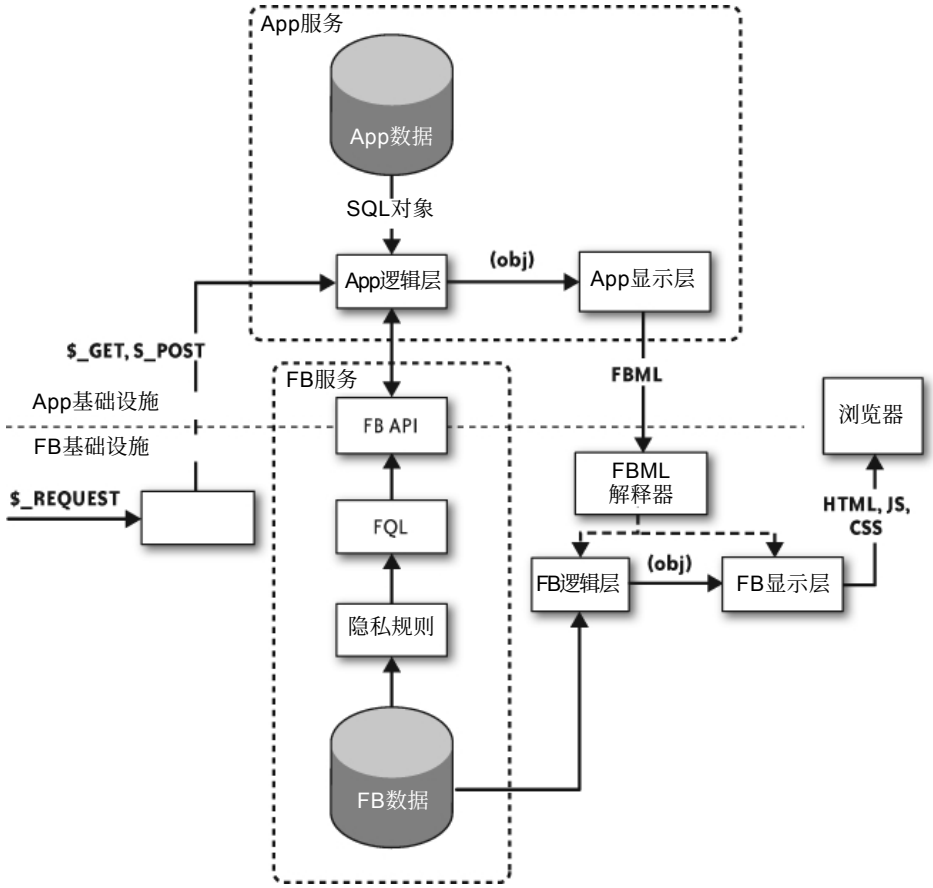

图6-5：应用即FBML服务

FBML是XML的一个特例，它包含了许多熟悉的HTML标签，增加了在Facebook上显示的平台专有的标签。FBML同样体现了FQL的高级模式：修改已知的标准（HTML，对FQL来说就是SQL），将执行和决定延迟到Facebook平台服务器上进行。如图6-5所示，FBML解释器让开发者通过FBML数据，自己能够控制在Facebook服务器上执行的逻辑和显示。这是数据处于执行中心的绝妙例子：FBML只是声明式的执行，而不是必须服从的控制流（如在C、PHP等语言中）。

现在来看具体细节。FBML是一个XML实例，所以它由标签、属性和内容组成。标签可以在概念上分成以下几大类。

直接的HTML标签

如果FBML服务返回标签<p/>，Facebook将在输出页上直接渲染为<p/>。作为Web展现的基石，大多数HTML标签都是支持的，少数违反Facebook层面的信任或设计期望的标签除外。

所以FBML字串<h2>Hello, welcome to Fetterman's books!</h2>在渲染成HTML时，实际上是保持不变的。

数据显示标签

这里是体现数据威力的地方。假定个人简介照片不能转到其他站点。通过指定<fb:profile-pic uid="8055">，开发者就可以在他们的应用中显示更多的Facebook用户信息，同时不要求用户完全信任开发者，将这部分信息交给开发者处理。

例如：

```
<fb:profile-pic uid="8055" linked="true" />
```

翻译成FBML：

```
<a href="http://www.facebook.com/profile.php?id=8055"

 onclick="(new Image()).src = '/ajax/ct.php?app_id=...">

  <img uid="8055" linked="true" src="..." alt="Dave Fetterman" title="Dave Fetterman" />
</a>
```

注意：复杂的onclick属性在生成时会在Facebook页面显示中限制Javascript。

请注意，即使信息受到了保护，这些内容也不会返回到应用栈中，只是显示给用户看。在容器端执行使这些数据可以查看，但不要求将它们交给应用程序！

数据执行标签

作为使用隐藏数据的一个更好的例子，用户的隐私限制只能通过内部的can_see方法来访问，它是应用体验的一个重要部分，但不能通过数据服务从外面进行访问。利用<fb:-if-can-see>标签和其他类似的标签，应用可以通过属性来指定一个目标用户，这样只有当查看者能够看到目标用户的特定内容时，那些子元素才会渲染出来。因此，隐私数据本身不会暴露给应用，同时应用又能满足强制实现的隐私设置。

从这个角度来说，FBML是一个受信任的声明式执行环境，与C或PHP这样的必须服从的执行环境不同。严格来说，FBML不像这些语言那样是“图灵完备”的（例如，没有提供循环结构）。像HTML一样，除了树状遍历所隐含的状态外，在执行时不保存任何状态；例如，<fb:tab-item>只在<fb:tabs>之内有意义。但是，通过让受信任系统中的用户获得数据，FBML提供了大量功能，这些功能正是大多数开发者希望提供给他们的用户的。

FBML实际上有助于定义执行应用的逻辑和显示，同时又让应用可以在应用服务器上显示独特的内容。

只面向设计的标签

Facebook因其设计标准而受到称赞，许多开发者都选择以某种方式复用Facebook的设计元素，保持Facebook的观感。通常，他们是通过利用*http://facebook.com*的JavaScript和CSS来实现的，但FBML提供了类似“设计宏”的库，以更可控的方式来满足这种需求。

例如，Facebook应用已知的CSS类来，将输入`<fb:tabs>…/fb:tabs>`渲染成特定的tab标签结构，位于开发者页面的顶部。这些设计元素也可以包含执行语义，如`<fb:narrow>…</fb:narrow>`只有在这次执行显示用户简介框中的少数列时，会在FBML中渲染它的子内容。

例6-22展示了一些使用只面向设计的标签的FBML。

例6-22：只面向设计的FBML的例子

```
<fb:tabs>
<fb:tab-item href="http://apps.facebook.com/fettermansbooks/mybooks.php"
 title='My Books' selected='true'/>
<fb:tab-item href="http://apps.facebook.com/fettermansbooks/recent.php"
 title='Recent Reviews' />
</fb:tabs>
```

这将渲染为一组可视的tab标签，链接到对应用的内容，并使用了Facebook自己的HTML、CSS和Javascript包。

替代HTML标签

HTML造成了一些信任风险，但没有暴露数据，所以FBML中的替代标签只是修改或限制一组参数，如Flash自动播放。这不是所有显示平台都严格要求的，它们只是强制应用满足容器站点的默认显示行为。但是，随着多个应用发展成为一个生态系统，它们都反映容器站点的观感，这种修改就变得很重要了。

请看这个FBML例子：

```
<fb:flv src=http://fettermansbooks.com/newtitles.flv height="400"
  width="400" title="New Releases">
```

这会翻译成一段相当长的JavaScript，渲染一个视频播放组件，这个元素由Facebook控制，特意禁止了自动播放这样的行为。

“功能包”标签

某些Facebook FBML标签包含了整套的常见Facebook应用功能。`<fb:friend-selector>`创建了类型前置的朋友选择器软件包，常见于许多Facebook页面，包括

Facebook数据（朋友、主要网络）、CSS样式和针对键盘动作的JavaScript。像这样的标签让容器站点可以推广某些设计模式和应用间的公用元素，也让开发者能够快速实现他们想要的功能。

FBML：一个小例子

请回忆一下我们在创建假想的外部网站时，通过引入`friends.get`和`users.getInfo` API对原来的*http://fettermansbooks.com*代码实现改进。接下来我们将展示一个例子，看看FBML如何能够结合社会关系、私有业务逻辑和完全集成的应用的感觉。如果我们能够通过数据库调用`book_get_all_reviews($isbn)`获得一本书的全部书评，那么我们就可以将朋友数据、私有业务逻辑和“墙式”风格结合起来，利用FBML在容器站点上显示书评，代码如例6-23所示。

例6-23：利用FBML创建一个应用

```
// Wall-style social book reviews on Facebook
// FBML Tags used: <fb:profile-pic>, <fb:name>, <fb:if-can-see>,
<fb:wall>

// from section 1.3
$facebook_friend_uids = $facebook_client->api_client->friends_get();
foreach($facebook_friend_uids as $facebook_friend) {
  if ($books_user_id = books_user_id_from_facebook_id($facebook_friend))
    $book_site_friends[] = $books_user_id;

}

// a hypothesized mapping, returning
// books_uid -> book_review object
$all_reviewers = get_all_book_reviews($isbn);

$friend_reviewers = array_intersect($book_site_friends, array_keys($all_reviewers));

echo 'Friends' reviews:<br/>';
echo '<fb:wall>';

// put friends up top.
foreach ($friend_reviewers as $book_uid => $review) {
  echo '<fb:wallpost uid="'.$book_uid.'">';
  echo '(' . $review['score'] . ')' . $review['commentary'];
  echo '</fb:wallpost>';
  unset($all_reviewers[$book_uid]); // don't include in nonfriends below.
}

echo 'Other reviews:<br/>';

// only nonfriends remain.
foreach ($all_reviewers as $book_uid => $review) {
  echo '<fb:if-can-see uid="'.$book_uid.'">'; // defaults to 'search' visibility
  echo '<fb:wallpost uid="'.$book_uid.'">';
  echo '(' . $review['score'] . ')' . $review['commentary'];
```

```
  echo '</fb:wallpost>';
  echo '</fb:if-can-see>';
}

echo '</fb:wall>';
```

虽然这采用的是输出FBML的服务的形式，而不是输出HTML的Web调用，但一般流程是不变的。这里，Facebook数据让应用能够在无关的书评之前，显示更多的相关书评（朋友的书评），并且使用了FBML来显示结果，采用了Facebook上相应的隐私逻辑和设计元素。

6.4.4 FBML架构

将开发者提供的FBML翻译成显示在*http://facebook.com*上的HTML，需要一些技术和概念综合作用：将输入字符串解析成一棵句法树，将这棵树中的标签转换成内部方法调用，应用FBML语法规则，保持容器站点的约束。像FQL一样，这里我们将关注点主要放在FBML与平台数据的交互上，对其他的技术则不作详细探讨。FBML处理了一个复杂的问题，FBML的全部实现细节是相当多的 —— 我们省略的内容包括FBML的错误日志、为后来的渲染事先缓存内容的能力、表单提交结果的安全性签名等。

首先，看看解析FBML的低层问题。在继承了浏览器的某些角色的同时，Facebook也继承了它的一些问题。为了方便开发者，我们不要求提供的输入可以通过schema验证，甚至不要求是结构良好的XML —— 不封闭的HTML标签，如`<p>`（与XHTML不同，即`<p/>`）打破了输入必须作为真正的XML进行解析的假定。因为这一点，我们需要一种方法将输入的FBML字符串先转换成结构良好的句法树，包含标签、属性和内容。

为了做到这一点，我们采用了采用了一个开放源代码浏览器的一些代码。本章将这部分处理视为一个黑盒，所以我们现在假定，在接收到FBML并经过这样的处理流程后，我们得到了名为`FBMLNode`的树状结构，它让我们能够查询生成的句法树中任何节点的标签、属性键值对和原始内容，并能够递归查询子元素。

从最高的层面上看，我们可以注意到FBML出现在Facebook站点的所有地方：应用“画布”页面、新闻信号源的故事内容、个人简介框的内容等。每种上下文中或每种“风味”的FBML都定义了对输入的约束，例如，画布允许使用iframe，而个人简介框则不允许。很自然，因为FBML维护数据隐私的方式与API类似，所以执行上下文中必须包含查看用户的ID和生成该内容的应用ID。

所以，在我们真正开始有效使用FBML之前，先要看看环境的规则，它由`FBMLFlavor`类来封装，如例6-24所示。

例6-24：FBMLFlavor类

```
abstract class FBMLFlavor {

// constructor takes array containing user and application_id
  public function FBMLFlavor ($environment_array) { ... }
  public function check($category) {
    $method_name = 'allows_' . $category;
    if (method_exists($this,$method_name)) {
      $category_allowed = $this->$method_name();
    } else {
      $category_allowed = $this->_default();
    }
    if (!$category_allowed))
        throw new FBMLException('Forbidden tag category '.$category.' in
this flavor.');
  }
  protected abstract function _default();
}
```

下面是这个抽象类的一个子类，它对应于渲染FBML的页面或元素。例6-25是一个例子。

例6-25：FBMLFlavor类的一个子类

```
class ProfileBoxFBMLFlavor extends FBMLFlavor {
  protected function _default() { return true; }
  public function allows_redirect() { return false; }
  public function allows_iframes() { return false; }
  public allows_visible_to() { return $this->_default(); }
 // ...
}
```

这种风味类的设计很简单：它包含了隐私上下文（用户和应用），实现了检查方法，为稍后将展示的`FBMLImplementation`类中包含的丰富逻辑建立了规则。与平台API的实现层很像，这个实现类为服务提供了实际的逻辑的数据访问，其他的代码为这些方法提供了访问入口。每个Facebook特有的标签，如`<fb:TAG-NAME>`，将有一个对应的实现方法`fb_TAG_NAME`（例如，类方法`fb_profile_pic`将实现`<fb:profile-pic>`标签的逻辑）。每个标准的HTML标签也都有一个对应的处理方法，名为`tag_TAG_NAME`。这些HTML处理方法通常让数据无变化地通过，但是即便是对一些“普通”的HTML元素，FBML常常也需要进行检查和转换。

让我们来看看某些标签的实现，然后将它们结合起来讨论。每个实现方法都接收一个来自FBML解析器的`FBMLNode`，以字符串的方式返回输出的HTML。下面是一些直接的HTML标签、数据显示标签和数据执行标签的实现示例。请注意，这些程序清单用到了一些功能，在这里没有完整而详细地列出。

在FBML中实现直接的HTML标签

例6-26包含了`<img>`标签的内部FBML实现。图像标签的实现包含更多的逻辑，有时候

需要将图像源的URL重写到Facebook服务器上图像缓存的URL。这体现了FBML的强大：应用栈可以返回与HTML非常相似的标记语言，支持它自己的站点，而Facebook可以通过纯技术的手段强制实现平台所要求的行为。

例6-26：fb:img标签的实现

```
class FBMLImplementation {
  public function __construct($flavor) {... }


  // <img>: example of direct HTML tag (section 4.3.1)
  public function tag_img($node) {

    // images are not allowed in some FBML contexts -
    // for example, the titles of feed stories
    $this->_flavor->check('images');

    // strip of transform attribute key-value pairs according to
    // rules in FBML
    $safe_attrs = $this->_html_rewriter->node_get_safe_attrs($node);
    if (isset($safe_attrs['src'])) {
      // may here rewrite image source to one on a Facebook CDN
      $safe_attrs['src'] = $this->safe_image_url($safe_attrs['src']);
    }
    return $this->_html_rewriter->render_html_singleton_tag($node->
    get_tag_name(), $safe_attrs);
  }
}
```

在FBML中实现数据显示标签

例6-27展示了通过FBML使用Facebook数据的例子。`<fb:profile-pic>`用到了`uid`、`size`和`title`属性，将它们结合起来，根据内部数据产生HTML输出，并符合Facebook的标准。在这个例子中，输出是指定用户名的个人简单照片，链接到用户的个人简介页面，只在当查看者能看到这部分内容时才显示。这个功能也存在于`FBMLImplementation`类中。

例6-27：fb:profile-pic标签的实现

```
// <fb:profile-pic>: example of data-display tag
public function fb_profile_pic($node) {
  // profile-pic is certainly disallowed if images are disallowed
  $this->check('images');

  $viewing_user = $this->get_env('user');
  $uid = $node->attr_int('uid', 0, true);
  if (!is_user_id($uid))
    throw new FBMLRenderException('Invalid uid for fb:profile_pic ('.$uid .')');

  $size = $node->attr('size', "thumb");
  $size = $this->validate_image_size($size);

  if (can_see($viewing_user, $uid, 'user', 'pic')) {
```

```
    // this wraps user_get_info, which consumes the user's 'pic' data field
    $img_src = get_profile_image_src($uid, $size);
  } else {
    return '';
  }
  $attrs['src'] = $img_src;
  if (!isset($attrs['title'])) {
    // we can include the user name information here too.
    // again, this function would wrap internal user_get_info
    $attrs['title'] = id_get_name($id);
  }

  return $this->_html_renderer->render_html_singleton_tag('img', $attrs);
}
```

FBML中的数据执行标签

FBML解析的递归本质使得<fb:if-can-see>标签就像是标准的必须服从的控制流中的if语句一样，它是FBML实际控制执行的一个例子。这是FBML实现类中的另一个方法，例6-28列出了它的细节。

例6-28：fb:if-can-see标签的实现

```
// <fb:if-can-see>: example of data-execution tag
public function fb_if_can_see($node) {
  global $legal_what_values; // the legal attr values (profile, friends, wall, etc.)
  $uid = $node->attr_int('uid', 0, true);
  $what = $node->attr_raw('what', 'search'); // default is 'search' visibility
  if (!isset($legal_what_values[$what]))
    return ''; // unknown value? not visible

  $viewer = $this->get_env('user');
  $predicate = can_see($viewer, $uid, 'user', $what);
   return $this->render_if($node, $predicate); // handles the else case
for us
}

// helper for the fb_if family of functions
protected function render_if($node, $predicate) {
  if ($predicate) {
    return $this->render_children($node);
  } else {
    return $this->render_else($node);
  }
}

protected function render_else($node) {
  $html = '';
  foreach ($node->get_children() as $child) {
    if ($child->get_tag_name() == 'fb:else') {
      $html .= $child->render_children($this);
    }
  }
```

```
  return $html;
}

public function fb_else($ignored_node) { return ''; }
```

如果某对“观察者-目标”通过了can-see检查，引擎就会递归地渲染<fb:if-can-see>节点的子节点。否则，就会渲染可选标签<fb:else>子节点下的内容。请注意fb_if_can_see直接访问<fb:else>子节点的方式；如果<fb:else>出现在这样的一个“if风格”的FBML标签之外，标签和它的子标签就不会返回任何内容。所以，FBML不仅仅是一个简单的转换式例程，它会注意到文档的结构，因此可以包含条件控制流的元素。

结合在一起

前面讨论的每个功能，都需要注册为一个回调，在解析输入的FBML时使用。在Facebook（以及它的开放源代码平台实现中），这个“黑盒”解析器是用C写的PHP扩展，每个回调都存在于PHP树中。要完成这种高层控制流，我们必须向FBML解析引擎声明这些标签。和其他地方一样，出于简单性考虑，例6-29也是经过了大量编辑的。

例6-29：FBML主要求值流程

```
// As input to this flow:
// $fbml_impl - the implementation instantiated above
//  $fbml_from_callback  -  the  raw  FBML  string  created  by  the  external
application

// a list of "Direct HTML" tags
$html_special = $fbml_impl->get_special_html_tags();

 // a list of FBML-specific tags (<fb:FOO>)
$fbml_tags = $fbml_impl->get_all_fb_tag_names();

// attributes of all tags to rewrite specially
$rewrite_attrs = array('onfocus', 'onclick', /* ... */);

// this defines the tag groups passed to flavor's check() function
// (e.g. 'images', 'bold', 'flash', 'forms', etc.)
$fbml_schema = schema_get_schema();

// Send the constraints and callback method names along
// to the internal C FBML parser.
fbml_complex_expand_tag_list_11($fbml_tags, $fbml_attrs,
  $html_special,$rewrite_attrs, $fbml_schema);

$parse_tree = fbml_parse_opaque_11($fbml_from_callback);
$fbml_tree = new FBMLNode($parse_tree['root']);

$html = $fbml_tree->render_html($fbml_impl);
```

FBML利用回调扩展了浏览器的解析技术，包装了由Facebook创建和管理的数据、执行和展现宏。这个简单的思想实现了应用的完全集成，支持使用通过API暴露出来的内部数据，

同时保持安全性方面的用户体验。FBML本身几乎就是一种编程语言，它也是充分发展后的数据：外部提供的声明式执行，安全地控制了Facebook上的数据、执行和显示。

6.5 系统的支持功能

现在，开发者创建的软件运行在Facebook的服务之上，不仅是结合了界面组件，而是全部的应用。在这个过程中，我们创造了一个社会关系网络应用的完全不同的概念。我们从一个典型的Web应用的独立数据、逻辑和显示的标准设置开始，不考虑所有社会关系数据，只是让用户可以确信能够作出贡献。现在，我们取得了充分的进展，应用使用了Facebook的社会关系数据服务，同时它自己又成为一个FBML服务，完全集成到容器站点之中。

Facebook数据也获得了长足的发展，不再仅仅是本章第一节讨论的内部库。但是，仍有一些重要的、常见的Web使用场景和技术，目前平台还未能支持。通过将应用变成一个返回FBML的服务，而不是直接由浏览器解读的HTML/CSS/JS，我们接触到了关于现代Web应用的一些重要假定。让我们来看看Facebook平台如何修正这样一些问题。

6.5.1 平台cookie

应用的新Web架构排除了浏览器内建的一些技术，许多Web应用栈可能依赖于这些技术。可能其中最重要的一点是，过去浏览器用于保存用户与应用栈交互信息的cookie不再可以得到了，因为应用的目标消费者不再是浏览器，而是Facebook平台。

初看上去，伴随对应用栈的请求发送一些cookie似乎是一个不错的解决方案。但是，这些cookie的作用域现在是“ *http://facebook.com* ”，而实际上，cookie信息属于该应用领域所提供的用户体验。

解决方案是什么？让Facebook具有浏览器的职责，在Facebook自己的存储库中复制这种cookie功能。如果应用的FBML服务送回请求头，试图设置浏览器cookie，Facebook就保存这个cookie信息，以（`user, application_id`）对为主键。Facebook然后“ 重新创建 ”这些cookie，就像用户向这个应用栈发出后续请求时浏览器所做的一样。

这个解决方案很简单，在开发者从HTML栈方式转向FBML服务方式转变时，只需要很少的改变。请注意，当用户决定在这个应用提供的HTML栈上导航时，这种信息是不能使用的。另一方面，它可以有效地分离用户在Facebook上的应用体验和在应用的HTML站点上的应用体验。

6.5.2 FBJS

当应用栈作为一个FBML服务被使用，而不是直接由用户的浏览器来使用，Facebook就

没有机会执行浏览器端的脚本。直接返回未修改过的开发者提供的内容（一个不充分的解决方案，这在FBML小节的一开始就讨论过）可以解决这个问题，但它违反了Facebook在显示体验上所加的约束。例如，当加载用户的简介页面时，Facebook不希望在加载事件上触发一个弹出窗口。但是，限制所有的JavaScript会排除许多有用的功能，如AJAX或在不重新加载的情况下动态操作页面的内容。

相反，FBML在解释开发者提供的`<script>`树和其他页面元素的内容时会考虑到这些约束。在此之上，Facebook提供了一些JavaScript库，让这些场景容易实现，同时又受到控制。这些修改共同构成了Facebook的平台JavaScript仿真套件，称为FBJS，它通过以下几点，让应用既动态又安全：

- 重写FBML属性，确保实现虚拟文档范围。
- 延迟激活脚本内容，直到用户在页面或元素上发起动作时。
- 提供一些Facebook库，以受控的方式来实现常见的脚本使用场景。

很清楚，不是所有的实现自有平台的容器站点都需要这些修改，但FBJS向我们展示了几种解决方案，这样的新Web架构需要这些解决方案来绕过一些困难。我们在这里只展示了这些解决方案的一般思想，FBJS的许多部分还需要不断改进，与FBML和可扩展的专有JavaScript库进行融合。

首先，JavaScript通常可以访问包含它的文档的整个文档对象模型（DOM）树。但是在平台画布页面中，Facebook包含了许多它自己的元素，开发者不允许对它们进行修改。解决方案是什么？在用户提供的HTML元素和JavaScipt符号之前加上前缀，即应用的ID（如`app1234567`）。通过这种方式，在开发者的JavaScript中如果试图调用不允许调用的`alert()`函数，就会调用未定义的函数`app1234567_alert`，并且只有开发者自己提供的那部分文档的HTML可以被`document.getElementById`这样的JavaScript代码访问。

作为FBJS需要对提供的FBML（包括`<script>`元素）进行这种转换的一个例子，我们创建了一个简单的FBML页面，实现了AJAX功能，如例6-30所示。

例6-30：一个使用FBJS的FBML页面

```
These links demonstrate the Ajax object:
<br /><a href="#" onclick="do_ajax(Ajax.RAW); return false;">AJAX
Time!</a><br />
<div>
<span id="ajax1"></span>
</div>

<script>
function do_ajax(type) {
  var ajax = new Ajax(); // FBJS Ajax library.
  ajax.responseType = type;
```

```
  switch (type) {
   <!-- note FBJS's Ajax object also implements AJAX.JSON and AJAX.FBML,
omitted
  for brevity -->
    case Ajax.RAW: ajax.ondone = function(data) {
      document.getElementById('ajax1').setTextValue(data);
    };
    break;
  };

 ajax.post('http://www.fettermansbooks.com/testajax.php?t='+type);

}
</script>
```

FBML和我们的FBJS修改动作将这些输入转变成了例6-31中的HTML。这个例子中的NOTE注释指出了每种需要的转换，不是实际输出的一部分。

例6-31：HTML和JavaScript输出的例子

```
<!-- NOTE 1-->
<script type="text/javascript" src="http://static.ak.fbcdn.net/
.../js/fbml.js"></script>

<!-- Application's HTML -->
These links demonstrate the Ajax object:
<br>
<!-- NOTE 2 -->
<a href="#" onclick="fbjs_sandbox.instances.a1234567.bootstrap();
 return fbjs_dom.eventHandler.call(
[fbjs_dom.get_instance(this,1234567),function(a1234567_event) {
a1234567_do_ajax(a1234567_Ajax.RAW);
return false;
}
,1234567],new fbjs_event(event));return true">
AJAX Time!</a>
<br>

<div>

<span id="app1234567_ajax1" fbcontext="b7f9b437d9f7"></span><!-- NOTE 3
-->
</div>

<!-- Facebook-generated FBJS bootstrapping -->
<script type="text/javascript">
var app=new fbjs_sandbox(1234567);
app.validation_vars={ <!-- Omitted for clarity -->};
app.context='b7f9b437d9f7';
app.contextd=<!-- Omitted for clarity -->;
app.data={"user":8055,"installed":false,"loggedin":true};
app.bootstrap();
</script>
```

```
<!-- Application's script -->

<script type="text/javascript">
function a1234567_do_ajax(a1234567_type) { <!-- NOTE 3 -->

var a1234567_ajax = new a1234567_Ajax();<!-- NOTE 3 -->
 a1234567_ajax.responseType = a1234567_type;
 switch (a1234567_type) {
  case a1234567_Ajax.RAW:
a1234567_ajax.ondone = function(a1234567_data) {
a1234567_document.getElementById('ajax1').setTextValue(a1234567_data);
};
break;
};

<!-- NOTE 4 -->
a1234567_ajax.post('http://www.fettermansbooks.com/testajax.php?t='+a1234
567_type);
}
</script>
```

下面是这段代码中的NOTE的解释：

NOTE 1

Facebook需要包含它自己的特殊JavaScript，包括`fbjs_sandbox`的定义，目的是渲染开发者的脚本。

NOTE 2

还记得前面FBML初始化流程中的`$rewrite_attrs`元素吗？FBML会重写这个列表中的属性，变成Facebook特有的功能；这实际上是FBJS的一部分。所以这里的`onclick`会激活这个页面的其他元素，这些元素在用户执行这个动作之前是非激活的。

NOTE 3

请注意在HTML和脚本中的元素如何加上了该应用的应用ID作为前缀。这意味着开发者对`alert()`的调用将变成对`app1234567_alert()`的调用。如果Facebook的后台JavaScript在这个上下文中允许这个方法，它将最终转向执行`alert()`。如果不允许，这将是未定义的调用。类似地，这种加前缀的方式实际上为DOM树提供了命名空间，所以对该文档某些部分的改变只限于开发者定义的那些部分。类似的沙盒技术也允许开发者提供限制范围的CSS。

NOTE 4

Facebook提供了一些专门的JavaScript对象，如Ajax和Dialog，目的是支持（并且常常改进了）常见的使用场景。例如，通过`Ajax()`对象发出的请求实际上能获得FBML作为结果，所以它们被重定向到Facebook域的一个代理上，在这里Facebook完成在线的FBML到HTML的转换。

支持FBJS需要对FBML进行改动、专门的JavaScript和AJAX代理这样的服务器端组件，才能够绕过应用Web架构的一些限制，但结果是很强大的。开发者因此可以享受绝大多数的JavaScript功能（甚至改进了这些功能，如支持FBML的AJAX），而且平台确保了应用内容提供了用户在Facebook上期望的受控体验，这完全是通过技术手段来实现的。

6.5.3 服务改进小结

解决了新的*n*层社会关系应用的概念带来的剩下一些问题之后，我们又改进了服务架构，添加了COOKIE和FBJS等项，如图6-6所示。

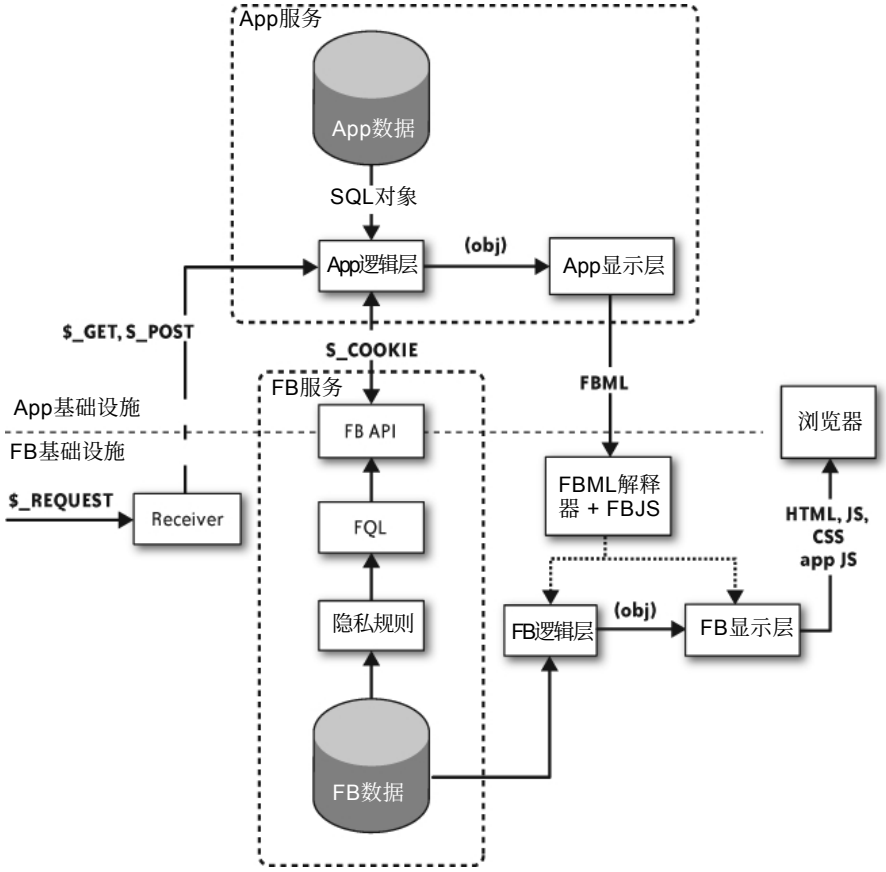

图6-6：Facebook平台服务

随着开发者的社会关系应用越来越成为Facebook使用的一项集成服务，而不是由浏览器使用的外部站点，我们已经重新创建或重新设计了浏览器的某些功能（通过平台cookie、FBJS等）。在试图改变或重建“ 应用 ”的概念时，这是必需的两个重要修改的例子。

Facebook平台包括类似的其他一些架构上的巧妙设计，这里没有详细介绍，其中包括数据存储API和浏览器端Web服务客户端。

6.6 总结

Facebook的用户贡献的社会关系有效地提高了*http://facebook.com*上几乎所有页面的效用。而且，这种数据非常通用，所以当它与外部开发者的应用栈结合在一起时，它的最佳使用就出现了，这都是通过Facebook平台的Web服务、数据查询服务和FBML等技术来实现的。从取得用户的朋友或简介信息的简单内部API开始，我们在本章中详细介绍的全部改进展示了如何协调不断扩展的数据访问方法和容器网站的预期，特别是对数据隐私和站点体验集成方面的要求。每次对数据架构的新改动都发现了Web架构的一些新问题，我们又通过对数据访问模式的更强改进来解决这些问题。

虽然我们将关注重点完全放在那些使用Facebook的社会关系数据平台的应用的潜力和约束上，但像这样的新型数据服务不一定局限于社会关系信息。随着用户贡献和使用的信息越来越多，这些信息在许多容器站点上都很有用（如内容收集、评论、位置信息、个人计划、协作等数据），各式各样的平台提供者可以应用Facebook平台特有的数据和Web架构背后的这些思想，并从中获益。