



NFC:

Arduino、Android与PhoneGap
近场通信

Beginning NFC: Near Field Communication with Arduino,
Android, and PhoneGap

[美] Tom Igoe, Don Coleman 著
Brian Jepson
金建刚 冯依 姚尚朗 译
极客学院 审校



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

版权信息

书名：NFC：Arduino、Android与PhoneGap近场通信
作者：（美）伊戈（Igoe,T.） （美）科尔曼（Coleman,D.）
（美）杰普森（Jepson,B.）
出版社：电子工业出版社
ISBN：9787121239977
定价：65.00

版权所有·侵权必究

O'Reilly Media, Inc.介绍

O'Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O'Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了Make杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项O'Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar博客有口皆碑。”

——Wired

“O'Reilly凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

本书谨献给Red Burns。

译者序

本书最开始是eoe社区CTO姚尚朗（iceskysl）推荐我们翻译的，很巧的是，之前我在上海的时候跟本书里面提到的Arduino有过一面之缘，并亲眼目睹过对其进行编程，然后运行出来之后的效果。真的感觉非常的酷，并且也非常容易上手，并没有想象中的那么神秘。恰好本书的内容跟Android也有非常大的联系，而NFC也是目前移动互联网中非常热门的技术。试看了一些章节之后，知道本书其实也没有涉及非常神秘、难懂的程序设计，或是一些其他门槛很高的内容。

本书并不是完全为程序员编写的，你可以是程序爱好者，也可以是经验老道的程序员；可以是做软件的，也可以是做相关硬件的；可以是Android开发者，也可以是iOS、黑莓、Windows Phone等；甚至可以是Web开发者。不管你使用什么语言进行开发，都能很快地上手。当然，如果你对我上面提到的技术都只是听过那也没关系，因为本书的门槛很低，只要你跟着本书一步步走，只要你对NFC感兴趣，想了解它的工作原理，以及知道能用它做些什么，那么本书绝对是一个非常好的选择。

本书从什么是NFC开始介绍，循序渐进地告诉我们如何去选择正确的、简单的工具来搭建一个简易的NFC应用。这里面还会涉及一些技术，比如HTML、JavaScript，如果你能完全了解当然最好，不过就算你只大概了解一些基本的标签、基本的语法也是没问题的；一旦你跟着本书开发出了一个简单的NFC应用，后续你想怎么去丰富它，我想那都是很容易的事情了。毕竟万事开头难。

在翻译本书的过程中，由于从广州来北京发展，所以翻译的进度有一些落后，特别感谢本书的翻译搭档hexter对我的帮助，在我由于其他事情而无暇顾及的时候帮我分担了一部分工作。也要感谢出版社编辑对本书的校稿，因为翻译其实是一件非常枯燥的事情，在翻译的过

程中难免会出现一些语句不通顺，甚至上下文不衔接的问题，编辑们得一行行地看，看不懂的需要标记出来，或者提供更好的句子。所以很感谢他们，让本书能顺利出版，然后为那些想了解NFC，并且想做出点什么的人提供一些好的入门材料。

第1章 简介

本书完完全全只是单纯的开始于2011年3月Brian写给Tom的一封邮件。当时我跟Brian正在编写另一本关于近场通信的书Making Things Talk，Brian觉得在这本书的第二版加入几行NFC的内容应该是个不错的想法，而且书中已经有一个关于射频识别（Radio Frequency Identification, RFID）的章节了，当时想这能有多难呢？两年半后，我们已经学了很多关于NFC方面的知识，并且在学习的过程中，遇到了一位非常优秀并且知识渊博的合作者Don Coleman，他是PhoneGap NFC插件的作者。

尽管NFC有很大的潜力，但是大部分书面材料都非常久远，并且都不是为非正式程序员而写的。所有的那些材料还假设如果你想知道关于NFC的东西，你就得准备从硅开始了解。你还必须明白不同类型RFID的细节，以及编写从NFC读取器中一次获取一个字节，然后编成字节流的代码。虽然这样做对于真正理解它非常有用，但是如果程序员能够更加专注地使用它，而不是关注一些底层的实现细节，那么NFC就能够看到更加广泛的应用范围了。当时Don（Don Coleman，这里简写为Don）的PhoneGap包是我们发现的最好的工具了，它能让你按照NFC论坛上设计师推荐的方式去设计NFC交互，你需要关注的只是交互信息，而不用担心其他的事情。

大部分这种书都是这样的宗旨：通过从设备到目标以及从设备到设备的读和写信息来学习NFC数据交互格式（NFC Data Exchange Format, NDEF）的基础，然后你会学到一些例子应用——有一些是为PhoneGap写的，有一些是为Arduino写的，还有一些是为Node.js写的——都是运行在类似于Raspberry Pi和BeagleBone Black等嵌入式设备上。你也会学习一些NDEF的用例，最后也可能会让你体验一下NFC驱动应用程序交互。

平台与平台之间有很多的技术变化，然而，NFC论坛的技术规范对于各个平台的非正式程序员来说并不是很容易接受的。我们已经尝试在这本书里给出一个路线图，尤其是在后面的章节中会谈到，目前的发展状况是什么样子，以及哪些地方还有改进的空间。

我们希望这本书能帮助非正式程序员对于使用NFC能够做什么有一个大概的了解，并且能激励专业的开发者创建更多简单易用的工具来帮助传播它的作用。

本书面向哪些读者

你不必是一个受过培训的专业程序员来读这本书，我们试着将本书为程序爱好者写出来——这样他们能获得一些知识，但不是一个正规的学习环境。你在这里不会学到编写企业级代码，但是你会得到实用的知识——关于什么是近场通信，以及如何编写程序并用在Android、Arduino和嵌入式Linux设备上。

我们假设你对编程有一些了解，你需要熟悉JavaScript和HTML，因为本书中有很多类似技术的例子。在Arduino项目中，你将需要懂得一点点C的知识，但是如果你熟悉JavaScript或者Java，也已经够了。对于后面的项目，你应该熟悉一点点电子产品，但是不是必需的。

推荐阅读

“什么？为了读这本书我还要去读其他的书？”不，没必要，但是我们在写这本书的时候发现一些其他很有用的书，我们想你可能会觉得它们一样有用。

如果你是JavaScript新手，则可以读Douglas Crockford的JavaScript: The Good Parts，然后认真思考一下；如果你是JavaScript老手，那么它会让你成为更好的程序员。他解释了语言的理论基础，并且有干净、简洁的好用例。

对于PhoneGap和Android，在线入门指南是最新的参考文献；详见PhoneGap开发门户（<http://phonegap.com/developer/>）和Android开发者网站（<http://developer.android.com/>）。想获取更深层次的Android介绍，详见Professional Android 4 Application Development或者Android Programming: The Big Nerd Ranch Guide。

想系统地获取更深层次的关于NFC的介绍，NFC论坛（<http://nfc-forum.org/ourwork/specifications-and-application-documents/specifications/nfc-forum-technicalspecifications/>）都有详细的原始素材（我们也转载了一些在附录A中作为参考）。我们也发现Vedat Coskun、Kerem Ok和Busra Ozdenizci写的Professional NFC Application Development也是很好的参考，尤其是对有经验的Java程序员。但在本书中我们采用了更加大众化的方式，因为我们的很多读者都是业余爱好者、黑客和其他自认为是程序员的“外行程序员”。

如果你是Arduino新手，Massimo Banzi的Getting Started with Arduino是一个非常好的起始点。Tom Lgoe的Making Things Talk, 2nd Edition也是一本对有经验的程序员学习关于将Arduino项目连接到网络的好书。Michael Margolis的Arduino Cookbook同样也能提供非常方便的方法去学习Arduino程序。

对于Node.js的介绍，会在本书的后面给出，Brett McLaughlin的What is Node? 是一个没有代码非常简洁的介绍。Manuel Kiessling的The Node Beginner Book以及Pedro Teixeira的Hands-On Node.js都是非常简短有用并且有真实代码，让你能立即上手的好书。

对于Raspberry Pi或者BeagleBone Black的很好的介绍，你可以看本书第9章，你也可以在Adafruit的指导材料(<http://learn.adafruit.com/>)中找到起始资料。Matt Richardson和Shawn Wallace编写的Getting Started with Raspberry Pi以及Matt Richardson编写的Getting Started with BeagleBone的书中都有非常好的介绍。

本书有哪些内容

第2章， NFC和RFID

本章通过对无线射频识别（RFID）介绍了近场通信（NFC）。简单地说，NFC是RFID的一个超集，它能做很多短距离内RFID能做的事情，等等，你会得到最重要的条件预览，看到一个NFC的系统架构并且学习到你需要哪些工具，到哪里去获得这些工具。

第3章， 从PhoneGap和PhoneGap库开始

本章介绍PhoneGap以及为PhoneGap编写的NFC插件。你将会安装一些必要的软件开始在Android上开发PhoneGap应用，编译和运行你第一次编写的例子。在本章的最后，你会使用Android设备阅读你的第一个NFC标签。

第4章， NDEF介绍

本章将会对NFC数据交换格式（NDEF）有一个深入的介绍，你会了解它的结构并且通过实践编写应用使用不同类型的NDEF记录执行相同的基本任务，看看每种记录类型是如何在Android上影响用户交互的。

第5章， 监听NDEF数据

本章涵养了如何在Android设备上监听NDEF信息，你将会学到如何过滤不同类型的标签和信息，以及开发NFC应用时非常有用的Android标签分发系统（Android Tag Dispatch system）。

第6章， 一个NFC应用实践

本章将创建一个完整的NFC应用（拥有完整的用户界面、音频播放和网络控制的灯光，所有的这些功能都是基于NFC标签来做

的) 运行在Android设备上。本章就是要告诉你如何策划交互设计和利用好NFC的应用数据格式。

第7章，Arduino和NFC介绍

本章带你进入另外一个平台——Arduino微控制器开发平台。你将学到如何使用NDEF包来读取和写入NDEF信息，也将使用Arduino和Node.js开发另外一个具有完整功能的应用。

第8章，点对点(P2P)交换

本章介绍在Android设备上使用NFC做P2P。你会学到通过P2P方式交流的数据记录类型是如何影响接收设备的，并且还将了解NFC是如何通过其他方式与其他载体交互的，比如蓝牙和WiFi。

第9章，嵌入式系统里的NFC

本章提供了嵌入式Linux平台的最新NFC开发例子（使用Raspberry Pi和BeagleBone）。你也将理解在嵌入式Linux设备上能做什么，并看到一些Node.js的例子程序。在此基础上它在可用性方面其实还有很大的改进空间，所以本章不是为了科技胆小者（不敢大胆创新去想NFC能够做更多的事情的人）而写的。如果你想从本章获得更大的收获，那么你需要对Linux命令行界面有一定的了解。这也是NFC中一些最让人兴奋的地方，所以是一个需要好好了解的范围。

你需要什么

为了做本书中的实例练习，你需要一些硬件和软件。幸运的是，所有的软件都是免费的。这里用的最贵的硬件是拥有NFC功能的Android设备。下面列出的就是你将会使用到的。

硬件

整体来说，根据本书你需要以下硬件：

- 一个具有NFC功能的Android设备
- 几个兼容NFC的标签（检查设备与标签的兼容性；“设备与标签类型匹配”章节中的表2-1将会告诉我们什么样的设备能匹配什么样的标签）

对于第6章，你需要：

- 一个Philips Hue照明灯系统 (<https://www.meethue.com/>)
- 一个蓝牙音乐接收器（例如，Belkin蓝牙音乐接收器 (<http://bit.ly/belkin-bmr>) 或者HomeSpot带有NFC功能的蓝牙音频接收器 (<http://www.amazon.com/HomeSpotNFC-enabled-Bluetooth-Receiver-System/dp/B009OBCAW2>)）

对于第7章，你需要：

- 一个Ardunio Uno微型控制器，很多地方有售，比如Ardunio (<http://store.arduino.cc/>)、Adafruit (<http://www.adafruit.com/>)、Seeed Studio (<http://www.seeedstudio.com/depot/>)、RadioShack等
- 一个NFC Shield（你可以使用Adafruit的Arduino PN532 NFC/RFID Controller Shield (<http://www.adafruit.com/products/789>)、Seeed Studio的NFC Shield (<http://bit.ly/seeed-shield>) 或NFC Sheild 2.0版 (<http://bit.ly/seeed-nfc-shield>)）
- 如果你使用的是Adafruit的Sheild，你可能会希望得到一些Adafruit的盾叠加头 (<https://www.adafruit.com/products/85>) 也一样
- 一个电磁驱动的门锁，12V或更小。我们使用的是在亚马逊买的埃美柯0837L直流12V 8W开放框架式电磁铁的电门锁 (<http://amzn.to/amico-lock>)，同时你也可以从其他零售商那里买螺线管。Adafruit卖的是一种类似的锁扣式螺线管

(<http://www.adafruit.com/products/1512>)，同时Seeed Studio也卖几种不同类型的，所以如果你从他们那里订购Shield，你还可以从那里买一个电磁阀

- 一个TIP120 Darlington（达林顿）晶体管 (<http://www.adafruit.com/products/976>)
- 一个12V、1000mA电源 (<http://www.adafruit.com/products/798>)，具有2.1mm内径，5.5mm外径，中心呈阳性连接器来驱动电磁阀电路
- Jumper线 (<http://www.adafruit.com/products/759>) 或22AWG实心线
- 两个LED灯（红，绿）。这些都可以从任何电子产品零售商那里买到，但仅供参考，请查阅Adafruit的红色LED组 (<http://www.adafruit.com/products/299>) 或绿色LED组 (<http://www.adafruit.com/products/298>)
- 两个220Ω电阻的LED

对于第9章，你需要：

- 一个BeagleBone黑色或Raspberry Pi的嵌入式Linux单片机
- 1A以上的电源（主板电源）
- 一个SLC3711非接触式的USB智能卡读取器 (<http://bit.ly/usb-reader>)
- 可选的，但有用：
 - 一个供电路板使用的USB WiFi适配器（Adafruit的微型无线（支持802.11b/g/n）
 - 模块 (<http://www.adafruit.com/products/814>) 效果很好)
 - 一个支持NFC适配器的A到A USB扩展器
 - 一个USB到TTL串行电缆——调试/控制台电缆 (<http://www.adafruit.com/products/954>)

如果你不想使用前面提到的设备，你可以在表1-1中找到一些我们经常使用到的电子分销商的零件版本号。

表1-1：本书中需要用到的电子元件

零件	MakerShed	Jameco	Digikey	SparkFun	Adafruit	Farnell	Arduino	Seeed
220Ω 电阻		690700	220QBK-ND			9337792		
							STR101C2M	
无焊接 电路板	MKEL3	20723	438-1045-ND	PRT-00137	64	4692810	或者	
							STR102C2M	
红色电线	MKSEED3	36856	C2117R-100-ND	PRT-08023		1662031		
黑色电线	MKSEED3	36792	C2117B-100-ND	PRT-08022		1662027		
蓝色电线	MKSEED3	36767				1662034		
12V、1000mA 直流电源（或其他 同等品）		170245		TOL-00298	798	636363		
Arduino Uno rev3 迷你控制 模块	MKSP11	2121105	1050-1017-ND	DEV-09950	50	1848687	A000046	ARD132D2P
绿色 LED 灯	MKEE7	333227		COM-09592		1334976		
红色 LED 灯	MKEE7	333973		COM-09590		2062463		
蓝色 LED 灯		2006764		COM-00529		1020554		
黄色 LED 灯	MKEE7	34825		COM-09594		1939531		
Mifare 射频识 别标签	MKPX4			SEN-10128				
NFC Shield	MKAD45				789		SLD01097P	
TIP120 Darlington (达林顿) 晶体管		10001 10001 32993_-1	TIP120-ND		976	9294210		
3.3V USB/ TTL 串口调试电缆				DEV-09717	954			
BeagleBone Black	MKCCE3	2176149	BB-BBLK-000- ND		1278	2291620	ARM00100P	
Raspberry Pi (B 型)	MKRPI2			DEV-11546	998	43W5302		

软件

从本书整体来看，你需要下列软件：

- Android软件开发工具包（详见第3章中的“设置开发环境”部分）
- Cordova CLI, PhoneGap工具包（详见第3章中的“为PhoneGap安装Cordova CLI”部分）
- Node.js和节点程序包管理器（NPM）
- 一个文本编辑器（我们喜欢作为跨平台GUI编辑器的Sublime Text2 (<http://www.sublimetext.com/2>)），不过你可以使用任何能生成一个纯文本文件的软件）

对于第6章，你会需要：

- Zepto jQuery包，可以从Zepto.js (<http://zeptojs.com/>) 中获得

对于第7章，你会需要：

- Arduino IDE (<http://bit.ly/arduino-software>)
- Seeed-Studio PN532包 (<https://github.com/Seeed-Studio/PN532>)
- Arduino NDEF包，可以从Don ColeMan的GitHub repository (<https://github.com/don/NDEF>) 获得
- Time包 (<http://bit.ly/time-library>)，是Michael Margolis为Arduino 开发的

现在不用担心如何去设置这些软件，后面我们会讲到。

其他不错的NFC应用

下面所列的应用对于整本书来说都是非常有用的。

- NXP（恩智浦）的NFC TagInfo (<http://bit.ly/nfc-taginfo>) 允许你读取任意NFC或者Mifare标签并检查上面的NDEF记录。
- NXP（恩智浦）的NFC TagWriter (<http://bit.ly/nfc-tagwriter>) 允许你做很多类似于TagInfo能做的事情，也可以写标签和非格式化标签，非常方便。
- NFC研究实验室的NFC TagInfo (<http://bit.ly/nfc-taginfo-nfcrl>) 会提供所有关于给定标签的信息，它比NXP的TagInfo更先进，它也允许你从标签中看到内存消耗，这在应用程序中是非常好的解决bug的方式。

对于第4章，你需要：

- TagStand的Trigger (<http://bit.ly/trigger-app>)
- TagStand的NFC Writer (<http://bit.ly/nfc-tagstand>)
- 三星公司的TecTiles（仅美国和加拿大可用）(<http://bit.ly/samsung-techtiles>)
- vvakame的App Lancher NFC Tag Writer (<http://bit.ly/app-lancher>)
- 供Android使用的一个Foursquare (<http://bit.ly/foursq-android>) 账户（如果你没有的话）

本书所使用的约定

以下是本书所使用的排版约定：

斜体 (Italic)

表示新的术语（中文则用楷体）、URL、电子邮件地址、文件名称和文件扩展名。

等宽字体 (Constant width)

用于表示程序片段，也可以用在正文中表示例如变量名或函数名等程序元素、数据库、数据类型、环境变量、语句和关键词（中文则用黑体）。



此图标代表提示、建议或者一般需要注意的。



此图标代表警告或者慎用。

使用代码示例

补充材料（代码示例、练习等）都能从
<https://github.com/tigoe/beginningnfc> 下载。

本书的目的就是为了帮助你完成工作，一般来说，本书提供的示例代码，除非你直接复制代码的重要部分，否则可以不用得到我们的许可而直接应用到你的程序或者文档中。例如，将本书中的几段代码放到你自己的程序中是不需要获得许可的。出售或分发来自O'Reilly图书的示例CD-ROM则必须得到许可。引用本书及本书中的示例代码来回答问题不需要得到许可。从本书将大量的示例代码整合到你的产品文档中则必须获得许可。

我们希望（但不是必须）你在发布自己的文档和代码时将引用出处写明。引用出处通常包括书名、作者、出版社和ISBN，例如：“Beginning NFC: Near Field Communication with Arduino, Android, and PhoneGap, by Tom Igoe, Don Coleman, and Brian Jepson. Copyright Tom Igoe, Don Coleman, and Brian Jepson 2014 978-1-4493-6307-9”。

如果觉得你使用的代码示例超出合理使用或上面给出的权限，请随时跟我们联系：

permissions@oreilly.com

Safari®联机丛书



Safari联机丛书是数字点播图书馆，它提供了在技术和业务上世界领先的作者的书籍和视频。

专业技术人员、软件开发人员、网页设计师以及商业人士和创意专业人士使用Safari联机丛书作为研究、解决问题、学习和认证培训的主要资源。

Safari联机丛书为组织、政府机构和个人提供了一系列产品组合和定价程序。用户可通过一个功能完备的数据库检索系统访问O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley&Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones&Bartlett, Course Technology,以及其他几十家出版社的成千上万种图书、培训视频和正式出版前的手稿。有关Safari联机丛书的详细信息，请访问我们的网站。

联系我们

关于本书的意见和问题请联系以下地址：

美国

O'Reilly Media, Inc.

1005 Gravenstein Highway North Sebastopol, CA 95472

800-998-9938 (美国或加拿大)

707-829-0515 (国际或本地)

707-829-0104 (传真)

中国

北京市西城区西直门南大街2号成铭大厦C座807室（100035）奥莱利技术咨询（北京）有限公司

我们为本书制作了一个Web页面，页面中包含了勘误表、示例以及其他信息。可以从这里访问这个页面：

<http://oreil.ly/beginning-nfc>

发表评论或询问有关本书的技术问题，请发电子邮件至：

bookquestions@oreilly.com

关于我们的图书、课程、会议和新闻的更多信息，请访问我们的网站：

<http://www.oreilly.com>

<http://www.oreilly.com.cn>

致谢

在本书的写作过程中，我们得到了很多人和组织的慷慨援助。PhoneGap的NFC插件是Kevin Griffin的心血，他和Don写了第一个版本并在2011年提交到PhoneGap上。Adafruit的Kevin Townsend一直是深入了解他的NXP软件和硬件最有价值的资源。Seeed Studio的Yihui Xiong和Seeed Arduino NDEF库是第7章成功的关键。Philippe Teuwen在第9章为libfreefare做的快速补丁也清理了很多障碍。Derek Molloy在BeagleBone Black的页面 (<http://derekmolloy.ie/beaglebone/>) 也是一个非常方便的资源。非常感谢Jody Culkin和Fritzing.org为本书提供的图片。早期的读者Ben Light、Sae Huh、Gabrielle Levine、Alex Kauffmann、Fil Maj和Dominick Gruntz也给我们提供了非常宝贵的意见。

本书之所以能成为可能，离不开我们的员工和纽约大学Interactive Telecommunications Program (ITP) 的同事们、Chariot Solutions公司、Maker Media以及Arduino的耐心支持。我们同时还要感谢一直支持我们的家人和合作伙伴。

第2章 NFC和RFID

近来，无线射频识别（RFID）技术正在成为日常生活的一部分。从轻触即付的支付卡，到收费公路上使用快易通E-ZPass标签卡的通过式收费系统，再到缝在商品上用来管理商品库存和阻止被盗窃的标签卡。生活中，我们大多数人一周会遇到好几次RFID标签的机会，但是，却很少会思考这种技术还能为我们做些什么。

在过去的几年中，一个与RFID紧密联系的新名词开始蓬勃发展，这就是近场通信（NFC）。问问那些普通的技术人员：NFC是什么？你可能会听到：哦，它和RFID一样，只是有点不同。但问题的关键是，咋个不同法？RFID和NFC常常混为一谈，但它们却是不相同的。尽管NFC读取器可以读写RFID标签，但NFC的能力却远超过RFID，而且有着更大的使用领域。你可以把NFC看作是RFID的一种扩展延伸，一个在众多RFID标准基础上创建的广泛的数据交换平台。

本书的目的就是向你介绍NFC技术及操作实战。通过章节中附带的练习实例，你将能够创建在具备NFC功能的Android设备和Arduino单片机上运行的NFC应用程序。你将了解到RFID和NFC有哪些功能是相同的，以及通过NFC能做些什么。

什么是RFID

夜晚，想象着你正坐在门廊上，当你打开门廊的灯时，看到邻居正好经过你的房子，你能看到他并认出他正是因为灯光反射回了你的眼睛，让你能够识别出你的邻居。这就是一种无源RFID。无线电信号从一个RFID读取器发射到无源RFID标签，该标签吸收能量并“汇报”身份。

现在，想象一下你打开门廊的灯，而你的邻居看到你后也打开他家的门廊灯，这样你就可以看到他并和他打招呼。这就是有源RFID。它部署在比较大的范围，因为接收器有电源供电，因此可以主动发射无线电信号而不依赖吸收发送方的无线电能量来供电。

RFID很像这两个门廊，你和你的邻居相互都认识对方的脸，但仅限认识。通过RFID这种方式，我们无法实现交换任何有意义的信息。所以，RFID并不是通信技术；相反，它只是用于识别。RFID标签可以持有少量的数据，并且你可以从RFID读取器对其进行读写，但能读写的信息是微不足道的，1000字节或者更少。

什么是NFC

现在，想象着另一个邻居从你家附近经过，看到她，你邀请她在门廊聊会天。她欣然接受，你们坐在一起，聊了些彼此生活中的趣事，你们的关系更亲密了。你们愉快地畅谈了几分钟。这就是NFC。

NFC构建在RFID基础上，能支持各设备间更加复杂的通信。你仍然可以用NFC读取器来读无源RFID标签，也能将信息写入该RFID有限的内存。你也能用NFC将标准格式的数据写入某些类型的RFID标签，而无须考虑标签类型的差异。当然你也可以与其他NFC设备进行双向通信。NFC设备之间可以交换彼此的功能集，交换设备上的记录信息，并能通过某种方式进行长期通信。

例如，你点击具备NFC功能的手机和具备NFC功能的立体声耳机，让它们相互识别，相互交换功能集，当彼此知道都有WiFi功能时，它们就能通过WiFi交换证书并开始通信。然后，手机就能通过WiFi在无线立体声耳机中播放音乐了。那么，为什么电话不能通过NFC传输音频呢？这里有两个原因：首先，NFC故意设计为近距离连接，一般有效距离为10厘米或更短。这使得它能做到低功耗，并能在使用时有效地避免干扰设备内的其他无线电元件；其次，与WiFi、蓝牙及其他通信协议相比，NFC的通信速率相对较低。NFC并不是设计用来管理高速通信扩展接口的，而是用于传递短消息、交换证书和初始化连接。回想一下前面提到的门廊聊天，NFC就是你邀请邻居聊天的过程。如果你想要和邻居长时间聊天，你就应该邀请她到家里一边喝茶一边聊天，这就是WiFi、蓝牙等其他扩展通信方式了。

最让人兴奋的是，NFC允许设备间在不交换密码、无配对，以及其他更复杂步骤的认证下，就能进行一些复杂内容的通信和短指令操作。这意味着，当你和朋友想通过手机交换地址信息时，你可以把手机放在一起，然后通过点击手机就能完成。当你想用谷歌钱包支付时，也只需轻点手机即可，就好像拥有了一张具备RFID功能的信用卡。

当你使用NFC时，你的设备不允许其他设备访问整个NFC内存，而只允许交换一些基础内容。你可以控制什么内容能发送，什么内容不能，并控制给谁发送。

RFID如何工作

一次RFID交互过程涉及两个角色：目标方和发起方。发起方，一个标签读取器或标签读写器，开始交互时，发起方产生一个射频电磁场，然后监听这个电磁场中目标方的响应。目标方，一个标签，当它感应到发起方的电磁场时就反馈一个响应。目标方会响应一个唯一的标识号（UID）。RFID有两种通信模式：有源和无源（主动和被动）。无源RFID交互过程涉及读写器和无源标签。这类标签通过吸收读写器发射的电磁场来获取能量，通常只要一点点能量就足够给读取器一个反馈响应了。有源RFID交互过程中的目标方是一个有源设备。因为目标方有电源供电，使得有源RFID的交互作用范围更大。E-ZPass和其他交通标识系统用的都是有源RFID。

RFID标签里有一小块内存，通常小于1KB，发起方可以读取里面的数据，如果发起方是一个读写器，它也能把数据写入标签，这样你就可以用来存储少量信用卡相关的信息。例如，交通系统中的不停车收费使用的RFID卡，它们被用来保存充值卡的余额信息。然而，通常的RFID系统都有网络数据库，所以更常见的使用方式是在标签中只存储具体数据记录的索引UID，而在远程数据库中保存这个索引对应的详细记录信息。

RFID标准体系

与普遍的观点相反，现在还没有一个统一的普适的RFID协议或规范，而是有几十个。RFID标准由国际标准化组织（ISO）联合RFID市场上的几个主要厂家起草定标。ISO是一个协调机构，协调不同行业领域的竞争对手之间制订可互操作的标准规范，即使他们之间是竞争关系，这项技术也是可以一起工作的。不同的RFID标准，定义了所使用的无线电频率、数据传输率、数据格式等。其中的一些标准定义了一个单一的可互操作的协议栈层，在后续的NFC章节里你将看到。其他一些标准则定义了各种完全不同的应用场景。例如，ISO-11784标准最初是为动物跟踪而制订的，它工作在129~139.4kHz频率之间，其数据格式也被设计成适合描述被跟踪的动物。你还可以找到工作在125kHz范围的EM4100协议，它通常被用于感应卡，该协议规定只需要保存非常有限的数据信息，通常只是一个UID信息。而ISO-14443标准族被制订用于支付系统和智能卡系统，它们工作在13.56MHz频率，其数据格式被设计成非常适合加减和加密等运算。例如，在14443标准族中有几种不同的格式，包括Philips和NXP的Mifare（非接触式读卡器）、索尼Felica（电子钱包）和NXP DESFire智能卡。ISO-14443A标准是和NFC标准兼容的，所以在后面你还会看到更多的相关内容。

NFC如何运作

NFC可以看作是RFID的延伸。与RFID类似，NFC设备之间的交互也有发起方和目标方。然而，NFC可以做的不仅仅是交换UID和将数据读写到目标方。RFID和NFC之间最有趣的区别是NFC的目标方通常是可编程设备，如移动电话。这意味着，NFC目标方不仅仅只是发送内存里的静态数据，实际上每次交互时它都可以生成独特的内容反馈给发起方。例如，如果你使用NFC在两个手机之间交换地址数据，NFC目标方设备可以编程做到对哪些从未见过的发起方只提供有限的信息内容。

NFC设备有两种通信模式。如果发起方一直提供电磁场能量，而目标方始终由该电磁场能量供电，这被称为无源通信模式。如果发起方和目标方各自都有电源供电，这被称为有源通信模式。这和RFID的通信模式一样。

NFC设备有三种操作模式。它们可以作为读写器，对目标方进行读出或者写入。它们也可以作为卡模拟器，在另一个NFC或RFID设备的电磁场里模拟成RFID标签使用。它们还可以工作在点对点模式下，进行双向数据交换。

NFC数据交换格式（NDEF）

NFC设备和标签之间的数据交换使用NFC数据交换格式（NDEF）。这是一个你会听到很多次的术语。NDEF是NFC对RFID的关键升级之一。它是运行在所有NFC设备之间的通用数据格式，不管是底层的标签还是其他技术设备。每条NDEF消息都包含一条或多条NDEF记录，每条记录都有一个特定的记录类型字段、一个唯一的ID字段、一个长度字段和有效数据载荷字段。有如下几种知名的NDEF记录类型，NFC设备知道如何处理这些类型。

简单文本记录

这类记录包含任何你想要发送的文本字符串内容，文本内容里一般不包含目标设备的操作指令。记录里也包含有指示语言和编码方案的元数据（metadata）（例如UTF-8）。

URI

这类记录包含网络地址。NDEF目标设备接收到一个URI记录后将传给能显示此记录的应用程序，比如Web浏览器。

Smart Poster（智能海报）

这类记录包含的数据附加到海报上，给它提供更多的信息。它可以包含URI数据，也可以包含其他数据，比如关于这个海报的文本内容，你可以把该内容发送给朋友。目标设备接收到智能海报记录后可能会打开浏览器、短信收发程序或者电子邮件应用程序，具体打开哪个取决于消息的内容。

Signature（签名）

签名提供了一种技术手段来保证包含在NDEF记录里数据的来源可靠性。

你可以在NDEF消息里混搭记录类型，也可以每条消息只发送一条记录，任你选择。

NDEF是RFID和NFC之间的一个重要技术差异。NFC和许多RFID协议都工作在13.56MHz频率，但RFID标签数据不必格式化成NDEF。不同的RFID协议无法共享通用的数据格式。

可以把NDEF消息理解成像句子那样的段落和记录：一个段落是一个离散的信息块，其中包含一个或多个句子。一个句子是一个小的信息块，其中包含一个信息点。例如，你可以写一段信息，指出你将有一个生日派对，并给出了地址。类比成在NDEF消息里，用一条简单文本记录来描述生日派对事件，一条智能海报记录给出物理地址，而一条URI记录给出获取更多细节信息的网址。

NFC的体系结构

为了深入理解NFC，我们要在心里构建一个NFC的分层结构模型。它有这么几层结构。最底下是物理层，这层包括CPU和用于通信的射频器件等物理部件。在物理层上面是数据分组和传输层，然后是数据格式层，最后是应用程序层。图2-1显示了NFC协议栈的各个层。

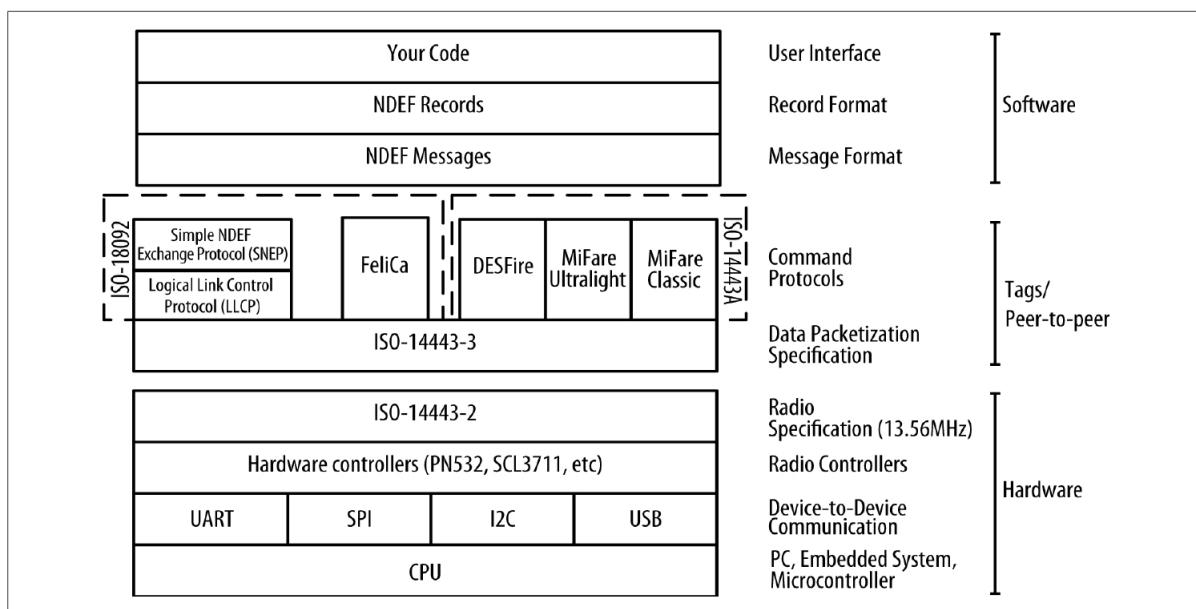


图2-1：NFC协议栈

在物理层，NFC采用了RFID的ISO-14443-2无线规范，它描述了13.56MHz频率的低功率无线电。接下来一层采用ISO-14443-3描述了射频发送数据的帧格式。任何你可能使用的无线电部件都是独立的硬件组件，不管是在手机或平板电脑里的，还是外插在单片机或个人电脑上的，它们采用一个或多个标准的设备间串行通信协议：通用异步收发器（UART）、串行外设接口（SPI）、内部集成电路通信（I2C）或通用串行总线（USB）与设备的主处理器进行通信。如果你是一个硬件发烧友，你可能知道前三种，但是如果你侧重于软件开发，你可能只知道USB。

以上提到的几种RFID命令协议，都基于两种规范：一种是构建在ISO-14443A上的RFID控制规范，NFC用此规范进行标签读写。

Philips/NXP半导体的Mifare Classic、Mifare Ultralight和NXP的DESFire协议兼容该规范；另一种是建立在ISO-18092基础上的NFC点对点交换控制规范。主要应用在日本市场的索尼FeliCa RFID卡和标签，也基于这个标准规范。即使你不用NFC，只要用这些标准协议规范也可以读写标签。

NFC和RFID之间的第一个主要区别是点对点通信模式，它采用ISO-18092标准实现。这里有两个协议：逻辑链路控制协议（LLCP）和管理点对点数据交换的简单NDEF交换协议（SNEP）。

RFID和NFC的第二个主要区别是基于这些控制协议的NFC数据交换格式（NDEF）。NDEF定义了由NDEF记录构成的消息数据交换。有几种不同的记录类型，在第4章你将了解更多信息。NDEF使应用程序代码可以使用相同的处理方式来处理NFC标签读写数据、点对点交换数据和卡模拟数据。

本书的大部分内容集中在NDEF消息的使用上。为了使你能更好地理解它们，我们也将讨论一些较低层的内容。我们认为NDEF的优点是，屏蔽底层差异性。

NFC标签类型

基于前面描述的RFID控制协议，NFC论坛定义了4种类型的标签。还有第5种兼容类型，但并不是严格遵守NFC规范。类型1、2和4都是基于ISO-14443A的，类型3基于ISO-18092。你可以从许多供应商处购得标签，它们的详细信息如下：

类型1

- 基于ISO-14443A规范。
- 可以是只读模式或读写模式。
- 内存空间为96B ~ 2KB。
- 通信速率为106Kbps。
- 没有数据碰撞保护机制。
- 例如：Innovidion的Topaz、Broadcom的BCM20203。

类型2

- 和类型1标签类似，类型2标签是基于NXP/Philips Mifare Ultralight 标签（ISO-14443A）规范的。
- 可以是只读模式或读写模式。内存空间为96B ~ 2KB。
- 通信速率为106Kbps。
- 有数据碰撞保护机制。
- 例如：NXP Mifare Ultralight。

类型3

- 这些是基于索尼FeliCa的标签（ISO-18092和JIS-X-6319-4），但不提供FeliCa支持的加密和认证功能。
- 工厂可配置只读或读写能力。
- 可变内存，每次交换最多1MB。
- 支持两种通信速率：212或424Kbps。
- 有数据碰撞保护机制。

- 例如：索尼FeliCa。

类型4

- 类似于类型1标签，类型4标签也是基于NXP DESFire标签（ISO-14443A）规范的。
- 工厂可配置只读或读写能力。
- 内存空间为2KB、4KB、8KB。
- 可变内存，每次交换最多32KB。
- 支持三种通信速率：106Kbps、212Kbps或424Kbps。
- 有数据碰撞保护机制。
- 例如：NXP DESFire、SmartMX-JCOP。

第5种类型是NXP半导体专有的类型，可能是如今最常用的NFC。

经典Mifare Classic标签（ISO-14443A）

- 内存选项：192B、768B或3584B。
- 通信速率为106Kbps。
- 有数据碰撞保护机制。
- 例如：NXP Mifare Classic1K、Mifare Classic 4K、Mifare Classic Mini。

从哪里买标签

Mifare和NFC标签越来越容易得到。为了充分利用这本书，你应该有一些NFC论坛标签和一些Mifare Classic标签为好。

现在很多手机商店出售三星的TecTile，它们适用于本书里的很多应用程序例子。虽然它们很贵，但使用起来很方便。最初的TecTile标签就是Mifare Classic，它在Arduino（见第7章）、USB NFC读取器、BeagleBone Black和Raspberry Pi（见第9章）上使用效果很好。目前尚不清楚，但看来三星正打算放弃TecTile而推崇TecTile 2。目前硬件厂商正在从Mifare Classic标签类型向NFC官方论坛标签类型转变。

如果你需要不同类型的标签，Adafruit有各种各样的NFC类型（<http://adafruit.com/category/55>）。SparkFun（<http://www.sparkfun.com>）和Seeed Studio（<http://www.seeedstudio.com>）也有大量的Mifare标签；搜索“NFC”或“Mifare”，你能搜到很多。

你还可以从TagAge（<http://www.tagage.net>）获得标签，但从芬兰发货可能会迟一点到你手上，这取决于你所在的位置。但他们的客户服务一直都非常好。

Identive NFC（<http://bit.ly/identivenfc>）可以在一夜之间交付标签。虽然他们的价格稍微贵了点，但到货很快。

设备与标签类型匹配

并不是所有NFC兼容的标签都适用于所有的NFC设备。在写这本书的过程中，我们测试了很多不同的设备和标签，发现一些组合能适配，而另一些则不能适配。虽然表2-1不是一个详尽的表，但提供它的目的是给你一些信息来适当地选择设备和标签。

你将碰到的最常见的问题是，有些设备能读Mifare Classic标签，而另一些则不能。Mifare Classic标签实际上并不是一个NFC论坛标准标签类型，最近，Broadcom和三星等主要硬件厂商已经开始不再支持在他们的设备上使用该类型标签。当你试图读取新版Mifare Classic标签时，很可能会得到不可预料的结果。例如，Nexus 4将Mifare Classic标签作为一种通用的标签来读取，而三星S4（谷歌版）则会显示一个错误说它不能处理该标签。如果可能，我们建议你使用官方NFC论坛标签类型。在后续章节中，我们会给出更具体的建议。

如果你学习NFC的原因是因为有兴趣使用它做嵌入式硬件项目，比如Arduino、Raspberry Pi或BeagleBone Black上的嵌入式项目更多信息见第7章和第9章，那么应该确保你有NFC标签和Mifare Classic标签。这些平台的开发库支持Mifare Classic标签。截至本书成稿时，这些库还不支持所有的NFC论坛标签类型。我们已增加了Arduino的NDEF库对Mifare Ultralight（NFC类型2）的阅读支持（不支持写入），并使得libfreefare库支持部分Ultralight和DESFire（NFC类型2和4）。libfreefare库的内容我们将在第9章详解。

虽然我们希望有额外的库来支持NFC论坛定义的所有标签类型，但可惜还没有。这是新兴技术发展之路上常见的烦恼。但到目前为止，有迹象表明该行业正朝着建立一套标准化的标签类型方向发展。

表2-1：设备与标签兼容表

Device	Type 1	Type 2 (Mifare Ultralight)	Type 3 (Sony FeliCa)	Type 4 (Mifare DESFire)	Mifare Classic
Samsung Galaxy S	Yes	Yes	Yes	Yes	Yes
Google Nexus S	Yes	Yes	Yes	Yes	Yes
Google Galaxy Nexus	Yes	Yes	Yes	Yes	Yes
Google Nexus 7 version 1	Yes	Yes	Yes	Yes	Yes
Google Nexus 7 version 2	Yes	Yes	Yes	Yes	No
Google Nexus 4 (phone)	Yes	Yes	Yes	Yes	No
Google Nexus 10 (tablet)	Yes	Yes	Yes	Yes	No
Samsung Galaxy S4 (tablet)	Yes	Yes	Yes	Yes	No
Samsung Galaxy SIII	Yes	Yes	Yes	Yes	Yes
Adafruit NFC Shield	No	Partial	No	No	Yes
Seeed Studio NFC Shield for Arduino	No	Partial	No	No	Yes
NFC USB Dongle (libnfc)	No	Partial	No	Partial	Yes

能用NFC做什么

现在有许多应用使用到了NFC。在货币交易方面，比如谷歌钱包、公共交通的支付和票务系统，都用到了NFC标签仿真功能。有几个手机应用程序，可以让你把手机的配置数据保存到标签里，并让你能用标签来自动改变手机的设置（例如，在开会时自动设为静音、点击标签自动打开WiFi并接入到预设的网络）。未来市场上的家庭音响设备，可自动将你的手机或平板电脑与音响或电视配对上，使前者作为遥控器能遥控后者。医疗保健系统里已经使用NFC技术使病人ID和病历记录对接，而且NFC论坛近期发布了个人健康设备通信

(PHDC) 技术规范用以在医疗领域发展NFC技术提供帮助。通过使用标签的NDEF自动记录货物的起运时间、运输时长、行程等信息，NFC技术使得库存管理效率得到极大的提高。在车队寻迹方面，通过使用NFC标签和读取器或点对点技术自动记录、反馈路由信息、行程时间、行驶里程、保养和油料等信息，使得寻迹管理能力大大增强。

在本书的写作过程中，我们已经在很多地方看到了NFC兼容标签的应用，从纽约市的Citi自行车钥匙环到中国的地铁公交卡再到挪威的酒店房间钥匙，等等。虽然这些应用并没有使用到标签上的全部NFC功能，但它们的使用都使原来的应用功能得到加强或简化。

通过组合点对点消息最小协商机制和设备标签通用数据格式功能，使得NFC在数据密集应用领域成为一个很有潜力的工具。

总结

近场通信（NFC）在RFID技术的基础上增加了一些有前景的新功能。其中最值得注意的是NFC数据交换格式（NDEF）。它提供了在4种不同NFC标签技术上的通用数据格式。NDEF既可用于标签到设备的数据交换，也可用于设备间的数据交换。NFC不仅仅是一种识别技术，它也是一种有用的短数据交换技术。NFC设备和标签之间的交互方式——只是一次点击——给用户一种很自然的交互体验。

有4种NFC标签类型，它们都是基于现有的RFID标准，其中前3种标签类型基于ISO14443A标准，第4种类型基于ISO-18092标准，这使得NFC标签至少部分兼容于许多现有的Mifare和FeliCa的RFID系统。尽管这些旧系统不支持NDEF，但它们仍然可以识别那些兼容它们的NFC标签。RFID读取器可以读取Mifare Ultralight标签。例如，虽然RFID读取器不能读取标签上的任何NDEF格式数据，但它仍能读出NFC类型2标签的UID。

当前，也有一些NFC设备不兼容旧的Mifare Classic标签。有些NFC射频支持非官方的第5种标签类型，而有些则不支持。随着NFC技术的成熟和更多的NFC设备出现在市场上，这些不兼容问题将会越来越少。但就目前而言，在你打算在应用程序中使用特定的读取器和标签之前，最好先检查一下它们之间的兼容性。

第3章 从PhoneGap和PhoneGap-NFC库开始

PhoneGap是一个开发框架，它允许你使用HTML5和JavaScript（在通常情况下）为iOS、Android、BlackBerry、Windows Phone 7和8、Symbian以及Bada开发应用。PhoneGap上的成员已经创建了一个本质上是浏览器的基本应用，但是没有界面。你要用HTML5和JavaScript实现界面，然后根据你的平台编译成相应的应用程序。他们还开发了一个系统插件，这样开发人员就能使用各个不同平台的原生功能在这个浏览器程序上进行拓展，这也意味着，你可以很方便地编写运行在不同平台上的应用程序，而且没必要知道不同平台的原生程序框架。

本书中，你将使用PhoneGap为Android以及那些允许你通过NFC硬件嵌入到不同Android手机中的插件编写应用程序。使用JavaScript编写用于监听标签的代码，而界面则使用HTML5呈现。应用程序shell则是使用Java编写的，由于是NFC插件，所以你没必要改变本书中的任何项目shell代码。

为什么选择Android

PhoneGap-NFC同时还支持Windows Phone 8、BlackBerry 7以及 BlackBerry 10。我们在本书中选择Android是因为它拥有最大的市场份额和最多支持NFC的手机，并且大多数PhoneGap-NFC插件用户已经成为了Android用户。如果你的是Windows Phone或者BlackBerry，你应该能够跟着Android的开发步骤修改本书中相应的例子。但是，有一些限制。Window Phone 8可以读写以及共享NFC标签，但是仅限于NDEF数据，它无法看到标签类型、标签UID或者其他标签元数据。BlackBerry 10也有类似的限制：它可以读和共享NFC标签，但目前无法写。

PhoneGap-NFC对BlackBerry 7的功能支持比BlackBerry 10的更多，自从PhoneGap 3.0对BlackBerry支持下降到BlackBerry 7之后，你就得使用旧版本的PhoneGap来支持了。你可以使用任何文本编辑器来编写代码，使用Android软件开发工具包（SDK）及其相关工具编译代码并部署到设备。你还需要了解Cordova命令行界面（CLI），它是使用Android SDK编译PhoneGap应用程序的工具集。同时你也需要了解一些计算机的命令行界面。

这些说明是假设你使用命令行界面工作，比如Linux或者OS X上的Terminal，对于Windows用户来说则是Command Prompt，这里面有一些细微的差别，我们也会指出的。

你的第一个PhoneGap应用：Hello，World！

对于本章中的项目，你需要：

- 一个文本编辑器（如果你没有最喜欢的文本编辑器，我们建议使用Sublime Text 2 (<http://www.sublimetext.com/2>)，因为它是一个很好的跨平台代码编辑器）
- 具有NFC功能的Android手机（见表2-1所示的可用的手机列表）
- 一些NFC标签（为了在多个设备上达到最佳效果，最好是使用NFC论坛上提到的类型，参见第2章中的“设备与标签类型匹配”一节查看更多的关于哪些设备与哪些标签能更好地协作）
- Android软件开发工具包
- Cordova CLI， PhoneGap工具箱（详见“为PhoneGap安装Cordova CLI”部分）
- Node.js和节点程序包管理器（NPM），用它来下载Cordova CLI（详见“安装Node.js和NPM”部分）

设置开发环境

首先，你需要从Android开发网站 (<http://bit.ly/sdkandroid>) 下载 Android SDK。Android建议是下载Android开发工具包（ADT），其中包括Eclipse开发环境的捆绑版本。但在本书中，我们不使用Eclipse，所以只需要为你的平台下载SDK就行了，这样比较节省时间。要做到这点，你只要在“下载（Download）”界面的底部点击“使用现有的IDE（Use an Existing IDE）”，就能得到一个只下载SDK（不包含Eclipse）的下载链接。

在OS X或Linux环境中

将Android SDK解压放在你能找到的电脑目录中。比如，在OS X系统中将它解压到Application文件夹中，那么Android SDK的目录则为/Applications/android-sdk-macosx。

在Windows环境中

你需要先在Oracle网站 (<http://java.oracle.com>) 上找到“Java SE”下载链接将JDK下载下来。安装好Java环境之后，就能运行Android SDK安装程序了。

在Windows上你还需要Apache Ant，它能用来管理Android程序的编译过程。从Apache Ant网站 (<http://bit.ly/apacheant>) 上下载Ant压缩文件，将其解压到C:\，并重命名顶层文件夹（如apache-ant-1.9.1-bin）。这样做完之后，你应该找到的子目录类似于C:\Ant\bin和C:\Ant\lib。



在Windows上，Android SDK安装程序会默认安装在home目录的隐藏目录下：C:

\Users\Username\AppData\Local\Android\android-sdk，这种方式会让你后期很难准确找到对应的目录，所以我们建议将其改为C:\Users\Username\Android\android-sdk。

安装Android平台工具

在OS X或Linux环境中

打开一个Terminal窗口，将目录切换到SDK目录（例如：使用cd/applications/Android-sdk-macosx）。该SDK没有配备任何版本的Android平台工具，所以你需要输入以下命令来安装：

```
$ tools/android update sdk --no-ui
```

在Windows环境中

如果Android SDK安装程序无法找到你的Java安装（64位Windows中的一个常见问题），那么就退出安装，并设置JAVA_HOME环境变量（在本节后面会指出）。再次尝试安装程序，它应该就能找到你的Java安装了。

安装完成后，你会得到安装工具的机会。只要选中“启动SDK管理器（Start SDK Manager）”复选框并点击“完成（Finish）”。随后SDK管理器就会出现，再点击“安装n包（Install n Packages）”（n是指默认情况下需要安装的包）。如果你没有选中“启动SDK管理器（Start SDK Manager）”复选框就直接点击了“完成（Finish）”，那么这时你可以打开Command Prompt，然后将目录切换到SDK目录（例如：cd Android\android-sdk）并运行下列命令：

```
C:\Users\Username\Android\android-sdk>tools\android update sdk --no-ui
```

这样将会更新Android SDK信息并安装最新版本的平台工具。在安装之前，可能还需要先接受软件安装协议。之后的安装过程会需要

一段时间，你可以去活动一下筋骨，喝点茶。接下来，你需要改变PATH环境变量来包含Android SDK工具，这个目录下的工具大部分用以编译、上传以及在设备上运行你的应用程序，在平台工具里，你会找到一个重要的工具叫adb，它是用来做后期调试的。



在下面的例子中，你应该根据自己的需要来改变路径。另外需要注意的是，你需要创建两条路径：一个是tools目录；另一个是platform tools目录。

在OS X或Linux环境中

将目录更改到home目录，然后编辑.bash_profile或.profile文件。如果文件不存在，你可以创建它，然后将下列语句添加到该文件中：

```
export ANDROID_HOME=/Applications/android-sdk-macosx
export PATH=$PATH:$ANDROID_HOME/tools:$ANDROID_HOME/platform-tools
```

如果你将SDK保存在Applications目录以外的路径，则改变ANDROID_HOME路径，然后保存profile文件，退出Terminal并重新登录设置PATH变量。

在Windows环境中

你必须找到“系统属性→高级”对话框：

Windows XP

打开开始菜单，右键单击“我的电脑”，选择“属性”，然后进入“高级”选项卡。

Windows 7或Vista

打开开始菜单，右键单击“计算机”，选择“属性”，然后从左边的列表中选择“高级系统设置”。

Windows 8.1

在桌面上，右键单击或轻触并按住开始菜单，选择“系统”，然后从左边的列表中选择“高级系统设置”。

点击“环境变量”按钮。在“用户变量”（不是“系统变量”）下面找到PATH条目，然后点击“编辑”按钮。如果没有PATH条目，则点击“新建”按钮，并命名新的变量PATH。

在后面加上下面三条路径，中间没有空格（最后两个条目能让你在命令行中运行Java和Ant）：

```
;%ANDROID_HOME%\tools ;  
%ANDROID_HOME%\platform-tools ;  
%JAVA_HOME%\bin ;
```



一定要保证路径之间有“；”来分隔（如果你已经创建了新的PATH变量，则可以跳过），不要改变变量里已经存在的东西，只能添加在最后面。如果不小心弄错了，则点击“取消”并重新试一次。不要修改“系统变量”，如果你修改了系统的PATH设置，你的系统可能会遇到很多问题。

在这个界面中，你还必须设置ANDROID_HOME和JAVA_HOME环境变量：

1.点击“确定”按钮关闭路径对话框，然后点击“用户变量”下的“新建”按钮。2.将名称设置为JAVA_HOME，值为C:\Program Files\Java\jdk1.7.0_25（或任何你安装JDK的目录）。

3.点击“确定”按钮关闭路径对话框，然后再次点击“新建”按钮。
4.将名称设置为ANDROID_HOME，值为C:\Users\%USERNAME%\Android\android-sdk。



如果你将Java和Android SDK安装在不同的目录下，那么改变这些环境变量是非常必要的。

最后，点击“确定”按钮，直到所有的对话框都关闭了。

安装Node.js和NPM

Node.js是使用JavaScript开发网络应用程序的平台，它也有一个非常好的系统安装包（就像PhoneGap一样），NPM（Node Package Manager）是为PhoneGap安装Cordova CLI最快的方式。你会在第7章和第9章的项目中看到Node的使用。

安装Node.js：

Windows

下载并安装Node.js (<http://nodejs.org>) 。



在安装之后，如果Windows命令行在运行Node时有任何的问题，你可能需要注销并再次登录查看PATH并更改。

Linux

使用包管理器（package manager）安装Node.js。如果NPM是Node.js中独立的包，则直接安装。例如，在Ubuntu上，你可以运行命令：sudo apt-get install nodejs npm。

Mac OS X

你可以使用网站上的Node.js安装程序或者使用类似于Homebrew (<http://brew.sh>) 的包管理器进行安装。如果安装程序没有将NPM模块添加到PATH，你可能需要在.bash_profile或者.profile文件中添加PATH，然后打开一个新的Terminal窗口：

```
export PATH=/usr/local/share/npm/bin:$PATH
```

你还需要安装Git。OS X和大多数Linux版本都会默认安装Git，如果没有，则使用包管理器进行安装。Windows上你也可以在得到Git命令行 (<http://git-scm.com/downloads>)，而Git安装程序会问你是否要将Git安装到系统的PATH环境变量中，我们建议你这样做，因为这可以让你直接在Command Prompt中运行Git。

为PhoneGap安装Cordova CLI

打开Terminal或者Command Prompt，然后安装Cordova包（这个包中安装的是PhoneGap框架）。

在OS X或Linux环境中

在Terminal prompt下运行以下命令：

```
$ npm install -g cordova
```

在Windows环境中

在Command Prompt窗口运行下面的命令：

```
> npm install -g cordova
```

-g选项会将Cordova命令行安装到PATH环境变量中的目录。

现在，你只需要输入下列命令就能创建PhoneGap项目了：

```
cordova create project-location package-name app-name
```

在上述命令中，project-location是Cordova Android项目的路径，package-name是应用程序的包名（使用反向域名的方式，例如com.example.myap），app-name则是项目名称。你还需要使用cd命令进入项目目录并添加Android平台：

```
cd /path/to/project-location  
cordova platform add android
```

在后面的章节中，你会经常看到这些步骤。对于本书，我们可以将上述步骤简单地说成“使用cordova create命令创建一个项目”。

现在，已经准备好创建你的第一个PhoneGap项目了！

PhoneGap还是Cordova?

PhoneGap和Cordova之间有什么区别? PhoneGap最初是由Nitobi开发的,后来被Adobe收购。在Nitobi被收购之前,他们将PhoneGap的代码库捐赠给了Apache软件基金会,以保证代码的开源管理工作。从Nitobi过渡到Apache时,这个项目的名字由PhoneGap改成了Cordova,PhoneGap现在是Adobe发布的Cordova版本。

从历史上看,PhoneGap和Cordova,以及PhoneGap背后的开源项目基本都是一样的。但在3.0版本时就改变了,现在Cordova的发布版本提供了一个将webview (browser) 嵌入到原生应用的核心功能。Cordova这个版本没有安装任何插件,这就可以让你只安装自己需要的功能,简化代码库,并减少应用程序的代码量。

PhoneGap是Cordova的分支,就像Safari和Chrome都是基于WebKit一样。Cordova的PhoneGap分支安装了所有的核心插件。此外,PhoneGap命令行还增加了类似于PhoneGap Build支持的功能选项。如果你没有安装原生SDK,那么该项目的编译可以委托给PhoneGap Build。

在本书中,我们交替地使用PhoneGap和Cordova。有关代码示例,我们使用了开源的Cordova版本。

创建PhoneGap项目

将目录切换到你想保存项目的任何路径（也许是你的文件目录），然后使用cordova create命令创建一个项目：

```
$ cordova create ~/Hello com.example.hello Hello
```

如果成功创建是没有输出信息的，但它会创建一个新的目录。这个命令将在你的工作目录下创建一个叫Hello的目录。将目录切换到它，并列出文件：

```
$ cd ~/Hello  
$ cordova platform add android  
$ ls
```

 如果Cordova需要你安装特定的Android版本（默认安装的除外），输入android启动SDK管理器，然后安装Cordova需要的SDK。另外，SDK的版本号可能和Cordova需要的对不上号（比如，Cordova可能提示需要Android 4.2 SDK，但是SDK管理器只提供4.2.2版本的），不过Cordova需要的Android版本是类似于API 17这种形式，在SDK管理器中列出的API版本号能对上，比如Android 4.2.2就是API 17）。在安装好Android版本之后，再使用cordova create命令重试一次。

在Windows上，命令会有点不同。打开Command Prompt（在默认情况下，它会处于你的home目录下，通常为C:\Users\username），不过可以肯定，你可能想先使用cd/D%UserProfile%（在大多数配置中，%USERPROFILE%环境变量包含home目录）命令来改变目录。当然，你也可以在其他的工作目录下运行以下命令：

```
> cd /D %UserProfile%  
> cordova create Hello com.example.hello Hello  
> cd Hello
```

```
> cordova platform add android  
> dir
```

在这三个平台中，目录中都会包含以下结构：

www

项目中的内容文件：图片、HTML文件等。这是你用得最多的地方。

platforms

同平台的支持文件，比如Android。

plug-ins

你安装的任何插件。

merges

在准备阶段特定平台的网络资源合并。

platforms/android目录下包含以下内容：

AndroidManifest.xml

这个文件描述了应用的结构、权限以及其他信息。

ant.properties

编译器属性文件。

assets

该项目主要目录www的副本。

bin

在编译过程中产生的二进制文件。

build.xml

该项目需要被编译的细节文件。

cordova

包含Cordova用来自动构建的各种脚本。

gen

在编译过程中自动生成的文件。

libs

任何额外需要的PhoneGap库文件，比如PhoneGap-NFC库。

local.properties

你的电脑上开发环境的属性。

proguard-project.txt

压缩和混淆代码的工具。

project.properties

项目的属性文件。

res

用来构建项目的资源，后期你将对这个目录下的文件进行修改。

src

应用程序的Java源文件，你没必要理会。

在Windows上，你可能需要为设备安装USB驱动。在早期更新SDK时，为不同的设备都会安装上USB驱动。然而，像Nexus 7这样的旗舰设备需要单独的驱动安装。例如，你可以从

ASUS (<http://bit.ly/1eMJkp0>) 下载 Nexus 7 的驱动。在 Android USB 驱动页面 (<http://bit.ly/usb-drivers>) 可以看到 Android SDK 驱动程序安装说明。

现在，你已经得到一个只有启动画面的应用了。编译并安装到你的设备上，使用 USB 连接设备，并运行以下命令（假设你现在使用的是 OS X 或 Linux）：

```
$ cordova run
```

如果是在 Windows 上，则运行以下命令：

```
> cordova run
```

当所有的东西都准备好后，你就能直接在 Android 手机或者平板上运行了。



在手机或平板上无法找到 Android 4.2 或以上版本的“开发者选项”，怎么办？打开“设置”并向下滚动到“关于设备”（在 Nexus 7 上是“关于平板”），点击“生成（Build）”7 次，这样就能启用“开发者选项”了。然后返回到上一个界面找到“开发者选项”并打开它。然后，你需要启用“保持唤醒”和“USB 调试”，如果你的设备有密码，则可能在编写代码时希望能禁用，因为这样就不用每次上传新的应用版本时都要重新解锁设备。

最后，打开手机并启动 Hello 应用程序，你应该可以看到类似于图 3-1 所示的界面。当你得到这个基本的“Hello, World！”程序后，试着修改 index.html，让它看起来更有意思。index.html 在 www 目录下面，添加一些按钮、图片和其他你想加进 HTML 页面的东西，然后保存文件重新运行，再看看你手机的变化。

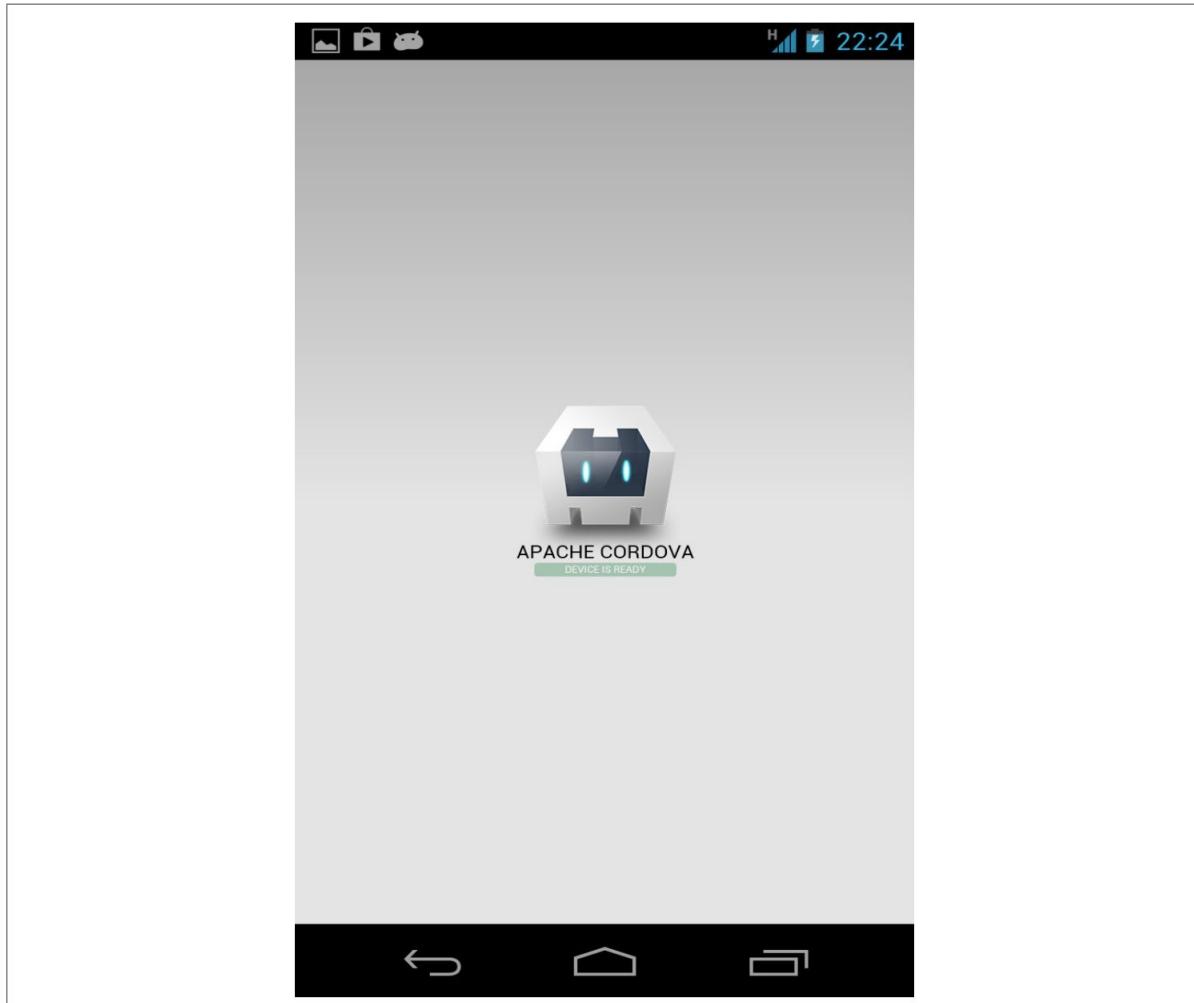


图3-1：PhoneGap HelloWorld应用程序

重要文件

在PhoneGap项目中有两个需要经常变动的文件：www目录下的index.html文件和www/js目录下的index.js文件。你可以添加其他的HTML和JavaScript文件，但是按照惯例，它们是程序的根，也是通过修改它们来使应用程序做你想做的事情。

index.html页面包含了应用程序的图形界面，就当它是可以放在任何网站的页面。index.js包含了事件句柄，比如onDeviceReady ()、initialize () 等，之后你还会添加一些函数到里面，比如当一个标签被阅读时，你可能添加一个onNfc () 函数。

你也可以通过创建一个新项目来得到未经修改过的默认文件结构。

还有一些你可能从来没有见到但是却影响应用的JavaScript文件：cordova.js、cordova_plugins.js以及cordova_plugins.json。cordova.js是PhoneGap库的主文件，cordova_plugins.js和cordova_plugins.json文件用来初始化你安装的插件，并提供入口到PhoneGap-NFC库和NFC硬件。当你安装插件和运行应用程序时，这些文件就会显示在platforms/android/assets/www目录中

JavaScript的习惯用语

如果你是JavaScript新手，PhoneGap默认的编码习惯或者编程模式可能让你觉得很怪。在很多JavaScript程序中，你可能见到很多初始化函数就像这样：

```
function toggleCompass() {  
}  
function init() {
```

```
}
```

如果你已经知道使用C或者Java编程，这样的模式可能让你觉得很熟悉，这些函数都是全局的，你可以在应用程序的任何地方像这样进行调用：

```
toggleCompass()
```

另外，在默认的Cordova index.js文件中，你会看到所有的函数都创建在一个叫作app的全局变量中。在JavaScript中，变量可以包含函数，你也可以将函数作为另外一个函数的参数。它们都是app对象的元素，被写成对象常量（object literal），就像：

```
var app = {
    initialize: function() {
    },
    bindEvents: function() {

    },
    onDeviceReady: function() {
    }
}
```

对象常量是一个用大括号围起来的键值对，由括号里的逗号分隔。如果你之前见过JavaScript Object Notation (JSON)，那么你就比较熟悉了。JSON是写对象常量的语法。它们仍然是函数，只是表现形式不一样，要调用这些函数，你只需要这样：

```
app.initialize();
app.bindEvents();
app.onDeviceReady();
```

将函数写在app对象里面意味着你知道它们都是本地函数，这样在复杂的应用中不会跟其他JavaScript库中的同名函数相冲突。

关于JavaScript的更多信息，可以查看Douglas Crockford的JavaScript: The Good Parts，它在JavaScript的结构和它如何工作方

面提供了非常好的介绍。

一个简单的定位应用程序

写完“Hello, World”之后，我们就能将它作为参考的模板，如果我们要做一个新的项目，“Hello, World”中包含的HTML和JavaScript都能拿来使用。为了使这个应用更加丰富，你可以将其扩展为通过点击来启动和停止跟踪经纬度的程序。



在以下的例子中，OS X、Linux（以\$开头的命令行）和Windows（以>开头的命令行）命令变量一个接着一个。在Windows上，大多数安装我们使用%UserProfile%指向home目录，但是你也可以根据自己的意愿使用其他的路径。

在以后的章节中，除非该命令是完全不同的，否则我们只给出OS X和Linux版本，Windows用户应该知道这之间的差异。

像之前那样，使用cordova create命令创建一个名为Locator的新项目：

```
$ cordova create ~/Locator com.example.locator Locator①
```

①Windows用户应该输入%UserProfile%\Locato来替代~/Locator。

然后将Android平台和地理位置插件加入到项目中，这些为PhoneGap/Cordova开发的插件继承了框架的功能。插件作者注册了插件的URL地址到Cordova项目，所以能被添加到插件数据库。很多插件都添加了到设备的硬件入口，像这个项目，就拥有获取设备地理位置系统的入口。你很快就能看到NFC插件就提供了到NFC无线电入口。后面你会看到插件是如何嵌入到文件结构中的，但是现在，这里只是演示如何安装地理位置插件。

```
$ cd ~/Locator①  
$ cordova platform add android
```

```
$ cordova plugin add \②  
https://git-wip-us.apache.org/repos/asf/cordova-plugin-geolocation.git
```

①Windows用户应该输入/d%UserProfile%\Locato来替代~/NfcReaderLocator。

②在Linux、OS X以及其他POSIX系统中表示续行字符。

现在将目录切换到项目的www目录，编辑index.html文件并删除其他一切东西。以下是一个非常小的HTML文件，它包含了在大多数应用程序中使用到的基本信息。里面有一个主要的名为app的div元素，所有的动作都将发生在这个div中。在index.js文件中你会看到它和它的子元素修饰相当多。在文件的末尾，包括cordova.js脚本和应用程序的index.js脚本。最后，从index.js中调用initialize()函数运行应用程序。

```
<!DOCTYPE html>  
  
<html>  
  <head>  
    <title>Locator</title>  
  </head>  
  <body style="font-size: 1.4em;">  
    <div class="app">  
      <div id="messageDiv"></div>  
    </div>  
    <script type="text/JavaScript" src="cordova.js"></script>  
    <script type="text/JavaScript" src="js/index.js"></script>  
    <script type="text/JavaScript">  
      app.initializeApp();  
    </script>  
  </body>  
</html>
```

编辑index.js文件并删除其他所有的东西，替换成下面的script代码。按照约定，所有的函数和变量在这个script中都是隶属于名为app的对象。通过初始化一个名为app的变量开始：

```
var app = {
```

```
};
```

接下来，添加需要初始化的一些功能，设置启动事件的监听器、按键触摸事件等。替换你刚刚添加的代码：

```
var app = {
/*
  Application constructor
*/
  initialize: function() {
    this.bindEvents();
    console.log("Starting Locator app");①
  },
/*
  bind any events that are required on startup to listeners:
*/
  bindEvents: function() {
    document.addEventListener('deviceready', this.onDeviceReady, false); ②
  },
/*
  this runs when the device is ready for user interaction:
*/
  onDeviceReady: function() {
    app.display("Locating...");
    app.watchLocation();
  },
};
```

①上述代码中的console是由浏览器提供的，现在每个浏览器基本都提供了JavaScript console视窗。

②上述代码中的document也是由浏览器提供的，代表当前你正在查看的HTML文件。

接下来，添加一个函数以获得设备的位置。这个函数有两个回调函数，一个处理成功结果，一个处理失败结果。如果成功，则调用display（）函数显示设备坐标；如果失败，则显示失败信息。

```
/*
  Displays the current position in the message div:
*/
  watchLocation: function() {
    // onSuccess Callback
    // This method accepts a `Position` object, which contains
    // the current GPS coordinates
```

```
function onSuccess(position) {
    app.clear();
    app.display('Latitude: ' + position.coords.latitude);
    app.display('Longitude: ' + position.coords.longitude);
    app.display(new Date().toString());
}

// onError Callback receives a PositionError object:
//
function onError(error) {
    app.display(error.message);
}

// Options: throw an error if no update is received every 30 seconds.
//
var watchId = navigator.geolocation.watchPosition(onSuccess, onError, {
    timeout: 30000,
    enableHighAccuracy: true
});
},
```

最后，需要将这些函数应用到用户界面。将下面两个函数 `display ()` 和 `clear ()`，写到 `index.html` 中的一个 HTML `div` 元素上。你会在本书的很多例子中看到这两个函数。

```
/*
    appends @message to the message div:
*/
display: function(message) {
    var label = document.createTextNode(message),
        lineBreak = document.createElement("br");
    messageDiv.appendChild(lineBreak);      // add a line break
    messageDiv.appendChild(label);         // add the text
},
/*
    clears the message div:
*/
clear: function() {
    messageDiv.innerHTML = "";
}
}; // end of app
```

本应用的完整源代码在 GitHub (<http://bit.ly/master-locator>) 上可以找到。

保存编辑后的文件，然后cd到项目的根目录，使用下列命令在手机上运行这个程序。

```
$ cd Locator  
$ cordova run  
  
> cd /d %userprofile%\Locator  
> cordova run
```

当在手机上运行这个程序时，你会得到定位应用程序显示的地理位置，如图3-2所示。

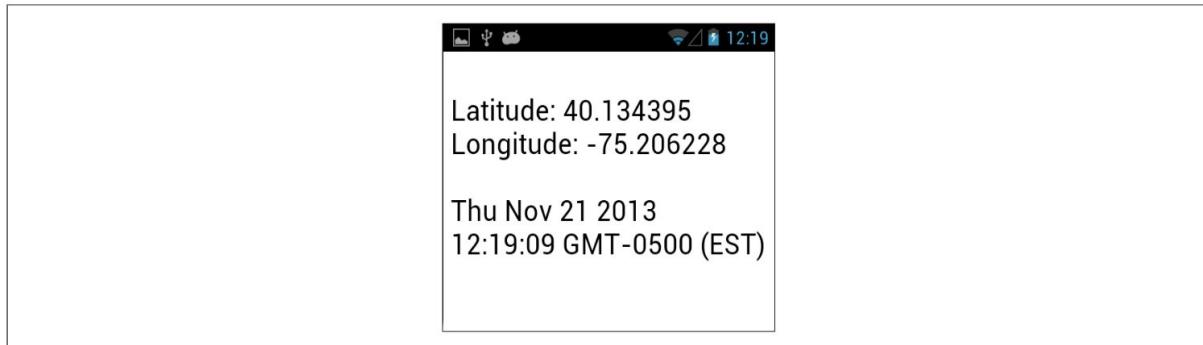


图3-2：定位应用程序显示的地理位置

你可以随意修改HTML、JavaScript，就像修改其他网页上的HTML、JavaScript或者CSS那样。一旦你明白了做一个PhoneGap应用程序的基本知识，你就可以继续前进了。

调试方式

有很多非常有用的工具可以用来调试PhoneGap程序，你可以选择使用对自己最有帮助的任何工具。

首先，你可能想要安装jshint，在运行代码之前用它来检查代码。安装jshint，需要输入：

```
$ npm install -g jshint
```

运行它，则输入：

```
$ jshint /path/to/js/file
```

当通过jshint运行代码时，它会在运行之前检查出错误或者其他格式错误。

如果设备还连接在电脑上，通过cordova run命令运行程序之后，输入adb logcat命令可以输出日志信息。

```
$ adb logcat
```

这个日志信息能显示Android手机上发生的一切事情，而且非常详细，甚至还包括JavaScript中的console.log信息。这些都非常的有用。例如，下面这些就是前面的应用程序输出的日志信息。

```
I/ActivityManager( 393): START u0 {act=android.intent.action.MAIN
    cat=[android.intent.category.LAUNCHER] flg=0x10200000
    cmp=com.example.locator/.Locator} from pid 590
D/dalvikvm( 2883): Late-enabling CheckJNI
I/ActivityManager( 393): Start proc com.example.locator for activity
    com.example.locator/.Locator: pid=2883 uid=10080
    gids={50080, 1006, 3003, 1015, 1028}

... twenty or so lines cut for brevity's sake ...
D/SoftKeyboardDetect( 2883): Ignore this event
I/ActivityManager( 393): Displayed com.example.locator/.Locator: +475ms
D/CordovaLog( 2883): file:///android_asset/www/js/index.js: Line 7 :
    Starting Locator app
I/Web Console( 2883): Starting Locator app at file:///android_asset/
    www/js/index.js:7
```

如果你想过滤掉除了网页上console信息之外的日志，则可以运行grep命令（适用于OS X或Linux）或findstr命令（适用于Windows），具体如下：

```
$ adb logcat | grep "Web Console"
```

```
> adb logcat | findstr /C:"Web Console"
```

这样做，在console视窗上就只显示JavaScript中的console.log（）信息，大部分时候它就只给出你需要关注的信息。下面是被过滤后的信息：

```
I/Web Console( 2335): Starting Locator app  
at file:///android_asset/www/js/index.js:7
```

你也可以通过同样的方式来过滤其他的信息，按“Ctrl+C”组合键可以停止显示。当然，如果你将设备与电脑断开连接，日志信息也会自动停止。

有些程序员倾向于在Terminal或者Command Prompt中运行应用程序，这样在编译和运行时可以不用总是启动或者停止应用程序（只需要观察控制台即可），其他人则喜欢使用启动或者停止的方式来运行程序，因为这样在每次运行程序时会有一个更加直观的感受。你可以试试两种不同的方式来找到它们之间的差异。

你也可以使用Andorid Debug Monitor（Android调试监视器）来查看运行情况。这是一个图形用户界面，位于Android SDK的tools子目录下。因为你之前已经加入了路径，所以这里你只需要输入以下命令就能启动它：

Linux或OS X：

```
$ monitor &
```

Windows：

```
> start monitor
```

这个命令会导致监视器在后台运行，在启动之后会返回到Command Prompt。如果你用的是Windows，监视器也会在后台启动，然后你就可以通过“Web Console”标签来过滤logcat的信息了，当然你也可以使用其他的日志标签，而且在监视器上还能看到其他的日志状态。还有性能监视器（performance monitor）、文件浏览器（file explorer）等。Android调试监视器的界面如图3-3所示。

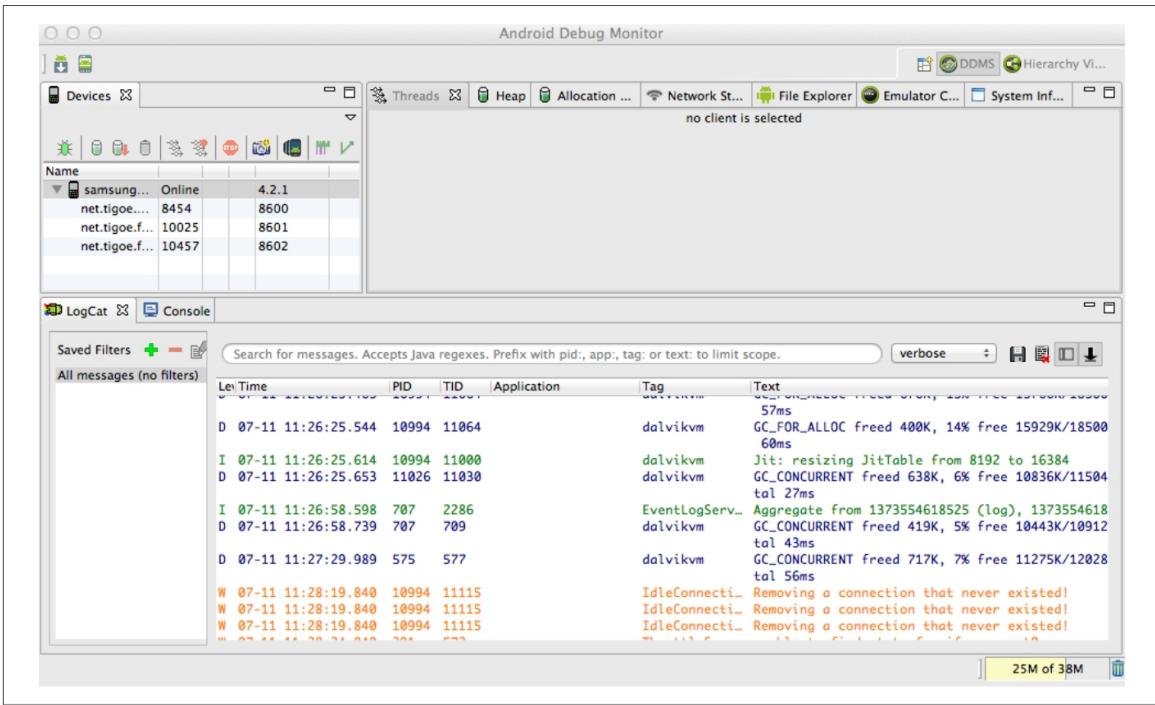


图3-3: Android调试监视器

如果你的设备连接到互联网，你同样可以在 <http://debug.phonegap.com> 上找到 JavaScript console 信息。只要在网页上填入一个类似于“kramnitz”的唯一名字，然后将这个 script 链接包含在 index.html 文件中，下次运行应用程序时就能在 <http://debug.phonegap.com/client/#kramnitz> 上找到 JavaScript console 了。非常的方便，但是必须得有网络连接。

debug.phonegap.com 有优点，也有缺点：你可以在上面看到 JavaScript 错误，但是有时你可能在看它从上面的 JavaScript 错误时忽略了 adb logcat 抓到的 Android 错误。而且如果运行一些负荷比较重的其他程序，debug.phonegap.com 运行有点慢。

NFC读取器

现在你对PhoneGap已经了解了，是时候去拓展它并让其实现NFC插件了。这次需要与第一个应用程序一样的软件、PhoneGap-NFC插件和一些NFC标签。第一个NFC应用程序只会读取任何标签的唯一ID。

之前用到的应用程序结构将用于所有的PhoneGap程序，所以记得保存好这个应用程序以便后期用来做项目模板。现在先创建一个名为NfcReader的程序。

```
$ cordova create ~/NfcReader com.example.nfcreader NfcReader ①
$ cd ~/NfcReader②
$ cordova platform add android
```

①Windows用户应该输入%UserProfile%\NfcReader来替代~/NfcReader。

②Windows用户应该输入/d%UserProfile%\NfcReader来替代~/NfcReader。

安装NFC插件

将NFC支持添加到项目，这与之前还是有一点不一样。下面的命令都会自动执行操作。

```
$ cordova plugin add /path/to/plugin
```

下面这个命令在全书中都会用到：

```
$ cordova plugin add https://github.com/chariotsolutions/phonegap-nfc.git
```

通过上面的命令，你可能也知道都安装了些什么：

- 插件就安装在项目的plugins/子目录下（在这个例子中是com.chariotsolutions.nfc.plugin），它也被安装在platforms/android/assets/www/plugins/下。
- 项目的AndroidManifest.xml文件和资源配置文件都需要修改，这样你的应用程序才能在设备上使用NFC读取器硬件。这个文件位于项目的platforms/android/目录下。
- 在platforms/android/res/xml/路径下的config.xml也需要修改。以下是两个文件修改的地方：

```
<feature name="NfcPlugin">
  <param name="android-package"
    value="com.chariotsolutions.nfc.plugin.NfcPlugin" />
</feature>
```

- 在AndroidManifest.xml文件中其他权限标签的末尾添加NFC的权限：

```
<uses-permission android:name="android.permission.NFC" />
<uses-feature android:name="android.hardware.nfc"
  android:required="false" />
```

这里有很多东西要记，不过只有一行的add插件命令能为你自动做好一切。

和前面一样，当你使用create命令创建好项目后，切换目录到项目的根目录，并如本节开头所示添加插件：

```
$ cd ~/NfcReader
$ cordova plugin add https://github.com/chariotsolutions/phonegap-nfc.git

> cd /d %userprofile%\NfcReader
> cordova plugin add https://github.com/chariotsolutions/phonegap-nfc.git
```

这个操作将会从在线资源库中将NFC插件安装到当前工作目录。

编写NFC读取器应用程序

现在你已经得到相应的资源了，是时候去修改程序本身了。在这个项目中有两个文件需要修改：

- www目录下的index.html文件。
- www/js目录下的index.js文件。

这个应用将监听NFC兼容的RFID标签并将它们的ID打印到屏幕上。

和前面一样，还是从index.html文件开始。与之前的定位应用程序的索引页面一样，index.html是一个包含了基本内容的页面。不同的是，在body元素的最后要添加一个Cordova脚本的调用，而且改变了body中app div元素，后者将会用来显示标签的ID。

打开index.html并替换成下面的代码，然后保存更改。

```
<!DOCTYPE html>

<html>
  <head>
    <title>NFC tag ID reader</title>
  </head>
  <body style="font-size: 1.5em">
    <div class="app">
      <div id="messageDiv"></div>
    </div>
    <script type="text/JavaScript" src="cordova.js"></script>
    <script type="text/JavaScript" src="js/index.js"></script>
    <script type="text/JavaScript">
      app.initialize();
    </script>
  </body>
</html>
```

index.js文件中有一个事件处理器用以监听NFC标签和将消息写到messageDiv中。像前面一样，需要先初始化应用程序变量并添加initialize () 和bindEvents () 函数。

```
var app = {
/*
  Application constructor
*/
  initialize: function() {
    this.bindEvents();
    console.log("Starting NFC Reader app");
  },

/*
  bind any events that are required on startup to listeners:
*/
  bindEvents: function() {
    document.addEventListener('deviceready', this.onDeviceReady, false);
  },
}
```

onDeviceReady () 对于现在来说还有一点复杂，当标签被NFC读取器发现时，你需要为其添加一个监听器。当NFC标签被读取时，nfc.addTagDiscoveredListener () 函数告诉NFC插件通知应用程序。第一个参数是当一个标签被扫描时调用的事件处理器；第二个和第三个参数是成功和失败回调插件的初始化。当成功时，将会调用onNfc () 函数；如果失败，则同样会给出失败信息。

addTagDiscoveredListener () 是NFC库中你能用的几个不同监听器中的一个，非常的通用，它不关心标签上有什么，只是在扫描到一个可以兼容的标签时发出通知。

```
/*
  this runs when the device is ready for user interaction:
*/
  onDeviceReady: function() {

    nfc.addTagDiscoveredListener(
      app.onNfc,          // tag successfully scanned
      function (status) { // listener successfully initialized
        app.display("Tap a tag to read its id number.");
      },
      function (error) { // listener fails to initialize

```

```
        app.display("NFC reader failed to initialize " +
            JSON.stringify(error));
    }
);
},

```

onNfc () 函数则在NFC事件中读取标签信息并通过display () 函数打印在屏幕上，就像你在定位应用中看到的一样，这些是应用里的最后几个函数。

```
/*
    displays tag ID from @nfcEvent in message div:
*/
onNfc: function(nfcEvent) {
    var tag = nfcEvent.tag;
    app.display("Read tag: " + nfc.bytesToHexString(tag.id));
},

/*
    appends @message to the message div:
*/
display: function(message) {
    var label = document.createTextNode(message),
        lineBreak = document.createElement("br");
    messageDiv.appendChild(lineBreak);      // add a line break
    messageDiv.appendChild(label);          // add the text
},
/*
    clears the message div:
*/
clear: function() {
    messageDiv.innerHTML = "";
}
}; // end of app

```

完整的应用源代码可以在GitHub (<http://bit.ly/tigoe-nfcreader>) 上找到。

上述就是所有需要改变的地方。保存好两个文件，如果没处于项目的根目录下，则切换到根目录，然后编译和安装程序：

```
$ cd NfcReader
$ cordova run
```

```
> cd /d %userprofile%\NfcReader  
> cordova run
```

输入命令之后，就将启动NFC读取器应用程序，当它运行时，将标签靠近手机，你应该会得到信息“Read tag”后面跟着一个UID（十六进制格式），如图3-4所示。如果你试着使用其他的标签，则会得到其他的UID。

项目托管

在你成功完成这个项目后，你想将它保存到Git版本仓库或者其他版本控制系统中，你可能不想存储一些构建文件、二进制文件等。删除这些文件或者配置Git的.gitignore文件，这样就不会将不必要的文件上传到版本仓库了。如果你已经像在“安装Node.js和NPM”一节描述的那样设置了PATH变量，那么可能都不再需要local.properties文件了。如果不需要，你就可以从远程仓库更新项目了，你可以通过以下命令来操作：

```
$ cd projectDirectory  
$ android update project -p platforms/android/
```

下面是你可以放心地从仓库忽略的列表，如果你正在使用Git，那么可以很简单地就将其.gitignore掉。

```
platforms/android/local.properties  
platforms/android/bin  
platforms/android/gen  
platforms/android/assets/www
```

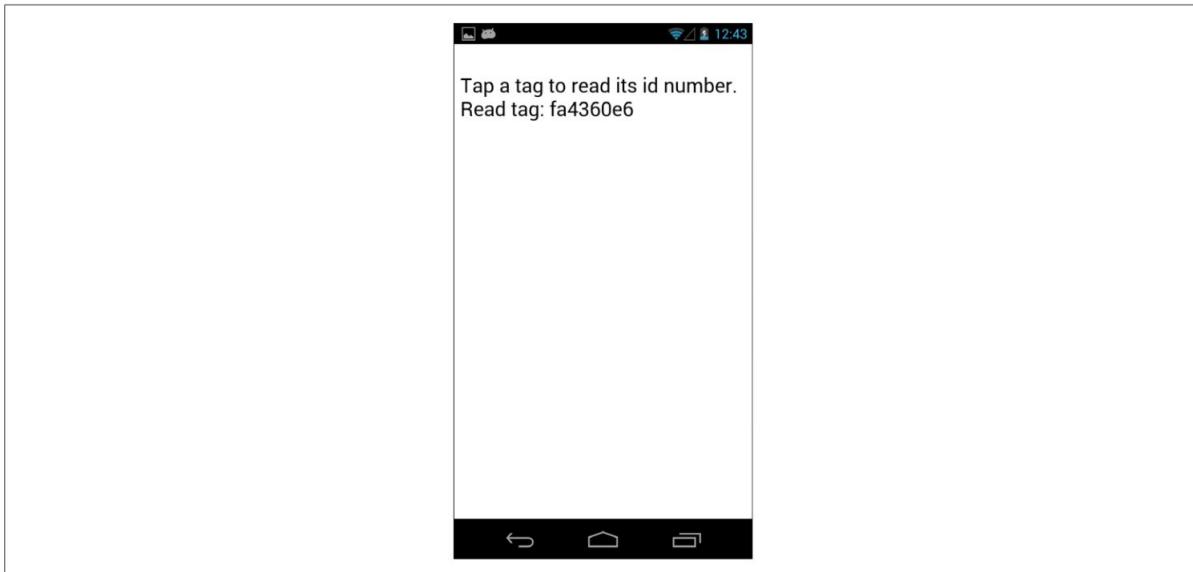


图3-4：NFC读取器应用程序读取一个标签

这些对于任何兼容NFC读取器的RFID标签都适用，这包括所有与ISO-14443A格式兼容的标签，包括Philips和NXP Mifare标签、Sony FeliCa标签、NXP DESFire。到目前为止，你还没有读取或者写入数据到标签，你只是读取了标签的唯一ID。不过，对于这个应用，从你开始将NDEF记录写入到标签中后，操作其他兼容的标签也就不成问题了。你会发现，这个程序可以读取Mifare Classic标签ID，甚至在那些与Mifare Classic理应不兼容的设备上。

现在已经证明设备上的读取器可以正常工作，并可以编写用来读取标签了。在下一章中，你将会学习使用NFC Data Exchange Format (NDEF) 如何从标签中读取数据以及将数据写入到标签。

故障排除

如果你的程序无法读取任何标签，则需要检查：

设备是否支持NFC功能？

打开设备上的“设置”并找到“更多”，如果你没有看到NFC，那么很不走运，你的设备不支持。如果你看到了，也要确保它能正常使用。当扫描到一个标签时，你应该得到一个确认的旋律，成功会是一个高声调的旋律，失败则是一个低声调的。如果NFC是禁用的或者不可用，PhoneGap-NFC插件会报告这个错误，这个应用会将这个信息打印到屏幕上。

使用的标签是兼容的吗？

当你在扫描标签时没有得到一个高声调的旋律，请确保它是一个兼容的标签，不然程序不会读取的。

设备接触好吗？

大多数设备的NFC读取器仅在设备背面部分，通常靠近上半部分。如果你没有读取到，试着将标签移到背面并找到读取器的位置。有时需要一两秒钟时间才能得到信息。

标签是不是损坏了？

破损的标签是无法工作的，或者是将标签与设备垂直放置了，请确保标签是平贴着设备的。

总结

本书中的PhoneGap-NFC项目都会参照本章的步骤，所以下面给出一个快速的摘要，以后可以做参考。

- 使用cordova create命令初始化一个通用的PhoneGap应用程序
- 使用cordova platform新增平台
- 使用cordova plugin添加PhoneGap-NFC资源

你的应用程序至少需要以下文件：

- index.html，在www目录下
- index.js，在www/js目录下

index.html文件需要包括下面的JavaScript脚本：

- cordova.js
- index.js

主要的JavaScript文件应包括PhoneGap-NFC库中的以下内容：

- 一个NFC监听器处理程序
- 一个响应NFC事件的函数

如果你已经懂得了所有的这些元素，那么就可以继续了。

到现在为止，你应该熟悉了PhoneGap应用程序的结构，并知道将PhoneGap-NFC插件添加到应用程序的必要步骤。在本书的其他项目中，你也会看到相同的结构。到目前为止，你所看到的只有RFID，还没有真正的NFC动作。但是到下一章，你将会学到NFC数据交换格式。

第4章 NDEF介绍

为了理解NFC，你需要了解NFC数据交换格式（NDEF），这是NFC设备和标签之间的语言。在本章中，你将学到NDEF的结构和它里面的记录。你还能写两个应用用于读取和写入NDEF格式的消息。

NDEF结构

NDEF是一种二进制格式的消息，每个消息可包含几条记录，如图4-1所示。每条记录都由报头和载荷构成。报头包含这条记录的元数据，如记录类型、长度等。载荷包含消息的具体内容。可以把NDEF消息想象成文章的一个段落，而记录就像其中的句子。一个组织良好的段落是由描述同一主题的句子组成的。同样的，NDEF消息是由描述同一主题的几条记录构成的，比如，地址簿条目。

NFC交互时间一般很短。每次交互通常只传送一个消息，每个标签也通常只携带一个消息。看一下NFC交互的物理场景：当你把设备和另一个设备或标签轻触时，整个信息交换过程只发生在你的设备与其他设备或标签接触的那一刻。如果要传送大数据量，在本书的最后一章会给出一个解决方案。但是现在，只考虑NFC每次交互只传送一个NDEF消息，而一个NDEF消息内含一条或多条记录。

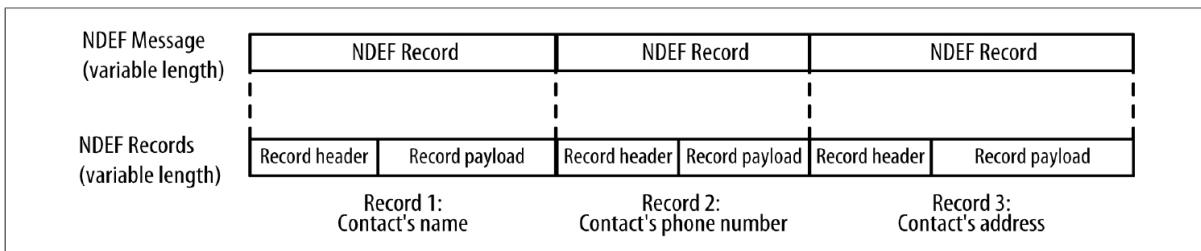


图4-1：NDEF消息由几条记录构成，这是一个典型的地址簿条目例子，有3条记录（姓名、电话号码、地址）

NDEF记录包含载荷数据和描述如何解析载荷的元数据。载荷有几种不同的数据类型。每条记录的报头都包含有描述这条记录的元数据和它在消息中的位置，还有记录的类型和ID。报头后面就是载荷。图4-2给出了一条NDEF记录的结构全貌。

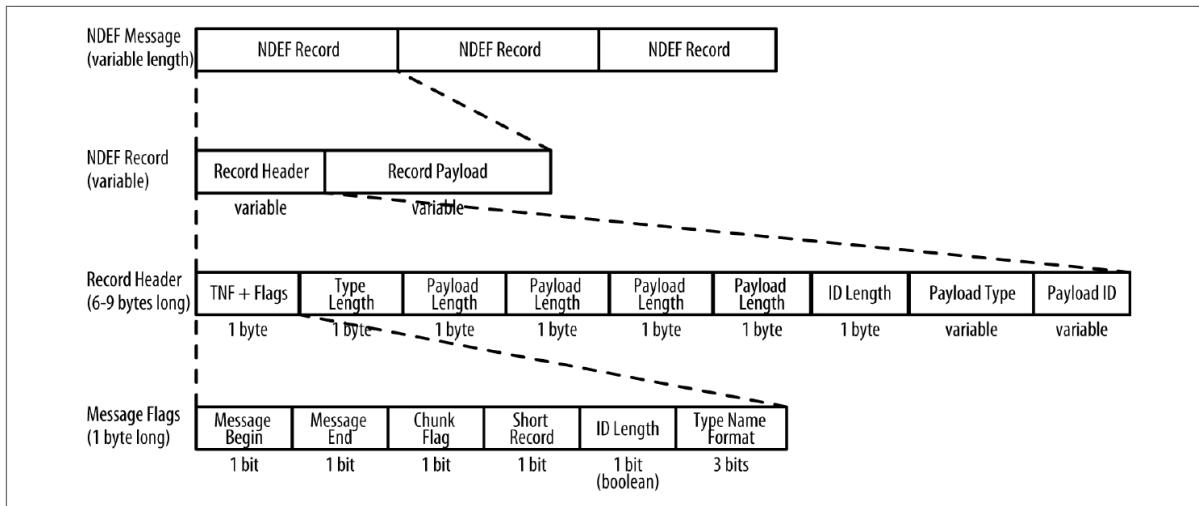


图4-2：NDEF消息结构

从图4-2可以看到一条NDEF记录由类型名称格式（TNF）、载荷类型、载荷ID和载荷构成。载荷是NDEF记录里最重要的部分，就是你要传送的内容。TNF告诉你如何解析载荷类型。载荷类型是NFC规定的类型：MIME媒体类型或URI。从另一个角度看，TNF就是载荷类型的元数据，而载荷类型就是载荷的元数据。载荷ID是可选的，并允许多个载荷通过ID关联或相互交叉引用。

TNF (类型名称格式)

每条NDEF记录都是以TNF开始的。TNF表明载荷类型字段值的结构。TNF告诉你如何解析载荷类型字段。可能有7种TNF值：

0 空 (NULL)

空记录，没有类型或载荷。

1 Well-Known

NFC论坛RTD规范预定义的类型。

2 MIME媒体类型

在RFC 2046中定义的互联网媒体类型。

3 绝对URI

在RFC 3986中定义URI类型。

4 外部 (External)

基于NFC论坛记录类型定义规范用户自定义的值。

5 未知 (Unknown)

类型未知，类型长度必须为0。

6 不变 (Unchanged)

仅仅表示分块载荷的中间记录和终止记录，类型长度必须为0。

7 保留

由NFC论坛留作将来使用。

在很多应用程序里，你可能会使用TNF 01（Well-Known）或TNF 02（MIME媒体类型）表示各种网络媒体。你也经常会看到TNF 04（外部），因为Android使用TNF 04触发应用程序打开。

载荷类型

载荷类型，也称为记录类型，更具体地描述了载荷的内容。TNF 定义了载荷类型的格式。类型可以是NDEF规定的类型、MIME媒体类型、URI或外部类型。NDEF记录类型定义（RTD）规范描述了一些 Well-Known记录类型，并设置了创建外部类型的规则。MIME RFC规范和URI RFC规范设置了其他类型的规则。

例如，一条TNF 01（Well-Known）记录的载荷类型可能是“T”（短信）、“U”（URI信息）或“Sp”（智能海报）。一条TNF 02（MIME媒体类型）记录的载荷类型可能是几种不同的类型之一，包括“text/html”、“text/json”和“image/gif”。一条TNF 03（绝对URI）记录表示将有一个文本URI，<http://schemas.xmlsoap.org/soap/envelope/>，作为类型。一条TNF 04（外部）记录也可能有几种不同的载荷类型，在本书中最常看到的是“android.com: pkg”类型。

欲了解更多的详细信息，请参阅NFC论坛记录类型定义的NDEF 规范文档（<http://bit.ly/specs-rtd>）。你也能在附录A中找到通用的NFC RTD规范列表。

NDEF消息可以包含多种载荷类型，但是按照惯例，第一条记录的载荷类型决定了整个消息的处理方式。例如，Android系统里对 NDEF消息的intent过滤处理只看第一条记录的载荷类型。

NDEF消息中的URI

在本书中你常会看到URI、URL和URN。URI或统一资源标识符是一个网络资源的字符串标识。URN或统一资源名称命名URI。URL 或统一资源定位告诉你得到网络资源所需的传输协议类型。如果URN 是你的名字，那么URI是你的地址，URL则告诉人们坐公共汽车到达你的地址。或者用网络术语：

URN

mySpecialApp

URI

net.tigoe.mySpecialApp (反转域名格式)

URL

<http://tigoe.net/mySpecialApp>

TNF的“绝对URI”有点误导。绝对URI意思是载荷类型是URI，而不是载荷是URI。而载荷类型字段里的URI则具体描述载荷，类似的有TNF 02的MIME类型字段描述载荷。例如：Windows和Windows Phone的LaunchApp消息记录使用TNF 03（绝对URI）类型，具体URI为“windows.com/LaunchApp”。LaunchApp消息提示用户启动一个应用程序，就像Android使用Android应用程序消息去启动应用程序一样。如果应用程序没有安装，则提示用户从电子商店下载。

Android系统处理绝对URI记录的方式有点违背NDEF规范。尽管NDEF规范规定了载荷类型描述载荷，但Android设备通过直接在浏览器中打开载荷类型字段的URI内容来处理TNF 03（绝对URI）记录。基本上，Android将URI载荷类型字段当作载荷来处理。而BlackBerry和Windows Phone扫描到绝对URI标签时则不打开浏览器。

如果你想发送一个URI或URL作为载荷，那么不应该使用TNF 03（绝对URI）。你应该把它们编码作为TNF 01（Well-Known）记录，使用NFC RTD“U”（URI）类型。NDEF规范提供了一个URI记录类型定义规范，它用URI ID码来编码URI，效率更高。例如，0x01表示<http://www>，0x02表示<https://www>。在“写不同的记录类型”一节你会看到一个例子，把URL添加到载荷，然后添加一个字节的0x01来作为<http://www>。在后续章节中你会看到这个应用程序更详细的内容，具体代码都在附录A中。

你也可以把URI编码到TNF 01（Well-Known）的NFC RTD“Sp”（智能海报）。使用智能海报记录允许在URI上附加额外信息，例如，多语种的文本描述、图标以及可选的处理指令。

载荷ID

载荷ID（载荷标识符）是一个可选字段，所填内容应该是一个有效的URI。它可以是一个相对URI，所以即使填“foo”也有效。应用程序通过该ID能识别记录内的载荷，也能使载荷引用其他载荷。由你决定载荷ID内容，但你也可以不用它。

载荷

载荷就是你的内容，它可以是通过字节流传输的任何你想要的东西。正常的NDEF库并不关心载荷的内容，只是传递它。你可以加密载荷，或者直接发送纯文本，你也可以发送一个二进制Blob或你能想到的其他任何东西。最终由发送应用程序和接收应用程序来商定载荷的意义和它的结构信息。

图4-2提供了记录头部第1个字节二进制格式的详细信息。你可能永远使用不到这些信息，因为任何编写良好的NDEF库都能为你处理它。即使你使用本书中的软件库，也无须考虑。如果你比较精通这些二进制格式信息，则可直接跳到“NDEF实战”一节。如果你想了解更多的消息和记录结构的知识，请继续看下一节。

记录结构

NDEF记录第1个字节的前5位是描述如何处理该记录和该记录位置的标志位。

这5个标志位如下：

MB（消息开始）

当这是消息中的第一条记录时为True，否则为False。

ME（消息结束）

当这是消息中的最后一条记录时为True，否则为False。

CF（块标志）

当这条记录是分块记录中的一条时为True，否则为False。

SR（短记录）

如果载荷长度字段使用短记录格式时为True，否则为False。。

IL（存在ID长度字段）

如果存在ID长度字段时为True，否则为False。。

注意IL不是ID字段的长度，它只是表明长度字段的存在。如果IL位为0，则表示没有ID长度字段或ID字段。

消息开始和消息结束标志位用于处理消息内的记录。因为NDEF消息是单条或多条NDEF记录的集合，并没有单纯的NDEF消息格式。根据MB和ME标志判定消息的开始和结束。消息中的第一条记录只有MB标志为true，中间记录的MB和ME这两个标志都为false，最后一条

记录只有ME标志为true。如果是一个只有一条记录的消息，则MB和ME标志都为true。

因为只有8种可能的载荷类型名称格式，所以只需3位来存储。TNF的位置在记录第一个字节的最后3位。

记录头部

NDEF记录是可变长度的数据结构。记录头部包含读取数据所需的信息。

一条NDEF记录开始于TNF字节，其中前5位就是前面讲的标志位。TNF字节之后，是一个字节的载荷类型长度字段，表示载荷类型字段的字节数。载荷类型长度字段是必需的，但值可以是0。

接下来的是载荷长度字段。SR标志位决定载荷长度字段的长度。如果SR标志位是true，则表示载荷长度字段是1个字节，否则是4个字节。载荷长度字段是必需的，但值可以是0。

如果IL标志位是true，则有ID长度字段，接下来的一个字节就是ID长度字段。

记录类型字段是一个长度可变字段，它在ID长度字段后（或载荷长度字段后，如果IL标志位是false）。类型长度字段决定应该读取多少字节的载荷类型字段。

如果有记录ID字段，那么它就在载荷类型字段之后。这个字段的字节长度由之前介绍的ID长度字段的数值决定。

记录头却结束，接下来的是载荷。

一个NDEF消息可以有多大

一条NDEF记录的载荷最大为 $2^{32}-1$ 字节（4GB-1），这是因为报头的载荷长度字段是4字节（32位）。但是，通过把记录串接在一起，可以形成更长的载荷。理论上，一个NDEF消息的长度没有限制。实际上，它受限于设备和标签的物理能力。如果只是进行设备之间的点对点通信，不涉及标签，那么NDEF消息大小只受限于设备的计算能力，和保持两个设备连接在一起的耐心。如果是进行设备和标签之间的通信，那么消息的大小受限于标签的内存容量。

使用NFC标签时，记录的大小是远低于 $2^{32}-1$ 字节的。NFC标签类型基于几种不同的RFID标签标准，大多数NFC标签类型是基于ISO-14443A标准的。这些标签的内存容量从96字节到4K字节不等，视标签类型不同而不同。Philips/NXP的Mifare系列标签与NFC兼容，包括Mifare Ultralight、Mifare Classic 1K和4K标签，以及Classic Mini。有一种基于日本工业标准（JIS）X6319-4的NFC标签可以达到最高1MB的内存，索尼的FeliCa标签就是这种类型的代表。关于标签类型规范的详细信息，请参阅NFC论坛网站的规范部分。

一般情况下，NFC通信是很快的。一个人拿着设备靠近标签或其他设备，一次简短的通信就发生了，随后她离开。NFC不是用于长期通信的协议，因为设备需要相互接触时才通信。当发送大消息时，用户必须将设备固定在合适的位置，直到消息传递完。这很单调乏味，所以人们通常使用NFC通信来交换设备的能力集，然后切换到WiFi或蓝牙进行大数据或媒体文件交换。

一个典型的NFC和WiFi协同工作的例子是：假设你有一个具有NFC功能的家庭音乐播放器，可以通过WiFi从中央家庭媒体服务器同步歌曲到你的智能手机或平板电脑。你正在听歌，但你必须出去工作。你轻触手机和播放器，播放器通过NFC告诉手机现在正在播放的是什么曲目，播放了多久。然后你的手机检查这首歌是否在其播放列表中，如果没有，它通过蜂窝网络或WiFi下载该歌曲。你出了门，戴上耳机，可以接着刚才的断点听这首歌。

记录组块

如果你需要发送超过 $2^{32}-1$ 字节的内容，则可以把载荷分解成块放入几条记录内传送。当这么操作时，你得设置组块标志（TNF的标志位）为true，第一分块和随后的所有分块都设置为true，除了最后一块记录设置为false。分块内容不能跨越多个NDEF消息分布。

第一分块记录的TNF设置后，随后分块的TNF设为06（不变）。中间和终止块必须设置类型长度字段为0。

每条记录的载荷长度表示载荷分块的大小。

超过500MB的消息通常不太可能出现，因此你用到分块功能的机会不是很多。但是，分块还可以用于传送动态生成的内容，尤其是在载荷长度事先不知道的场合。

分块很少使用，所以本书中使用的库没有实现分块。Android的解析器能读取分块信息并将它们组合成逻辑NDEF记录。

参考

更多关于NDEF结构的信息，包括一个编写自己的NDEF解析引擎的测试手册，请访问NFC论坛规范页面 (<http://bit.ly/nfc-tech-specs>)。

NDEF实战

在学习NDEF实战前，不妨先看看现有的应用程序是怎么做的。在这部分，你需要下载一些已有的应用程序到设备，以便可以写一些标签并比对它们的工作。

许多流行的标签应用最常用的功能之一就是Foursquare签到功能。当你把设备轻触有Foursquare功能的标签时，设备将自动连接到社交媒体Foursquare并签到该地点。然而，每个应用程序处理该签到功能略有差异，其差异反映在结果中。

对于这个实战项目，你需要：

- 一部具有NFC功能的Android手机
- 5个NFC标签
- 安装接下来列出的应用程序（你可以从Google Play将这些应用程序直接安装到设备）

希望已经读了第3章并安装了所需的所有软件，以完成这一章中的示例。



跨多个设备使用时，最好使用NFC论坛标签类型；Mifare Classic标签可能会与很多新上市的设备不兼容。参见“设备与标签类型匹配”一节获取设备与标签的兼容关系。避免使用类型2（Mifare Ultralight）标签，因为该类型标签没有足够的内存用于此实战项目。

下面是本次实战项目使用的应用程序：

- TagStand公司的Trigger (<http://bit.ly/trigger-app>)。
- NXP公司的NFC TagWriter (<http://bit.ly/nfc-tagwriter>)
- TagStand公司的NFC Writer (<http://bit.ly/nfc-tagstand>)

- 三星公司的TecTiles (<http://bit.ly/samsung-techtiles>) (仅美国和加拿大可用)
- vvakame公司的App Lancher NFC Tag Writer (<http://bit.ly/app-lancher>)

你还需要NXP的TagInfo (<http://bit.ly/nfc-taginfo>) 来读取标签内容、Android平台的Foursquare应用程序 (<http://bit.ly/foursq-android>) 和 Foursquare (<http://www.foursquare.com>) 账户。

在测试之前，请务必登录Foursquare应用程序。如果你遇到任何问题（如Foursquare应用程序闪退），请退出应用程序，并重新登录。有些问题完全超出你的控制。例如，设备把载荷移交到Android系统后，应用程序（如Foursquare）负责接下来的工作，如果该应用程序没法处理好这些数据，你可能会一头雾水，不知道问题出在哪儿。

就算签到信息不是你真正所在的位置也不用担心。首先，每个应用程序在实际执行签到之前都必须请求Foursquare的使用许可，此时你可以取消签到。其次，之后你随时可以在Foursquare网站检索和删除你的签到记录。在测试NFC时，当心你的朋友搞混了你的位置，那就糟糕了。

每个应用程序的写标签过程都相当简单，除了TagWriter。以下是基本步骤：

NFC Task Launcher

打开应用程序，并点击顶部的“+”开始一个新任务。从任务类别列表中选择“NFC”。命名任务，然后点击“+”按钮。从列表中选择“社交媒体”，然后选择“Foursquare签到地点”。通过键入一个名称选择签到地点，或者点击放大镜让应用程序帮你寻找附近的签到场地。点击“确定”按钮，然后点击“添加到任务”。点击右箭头写标签。把手机贴近标签，应用程序就写标签了。

TagStand Writer

打开软件，点击“Foursquare场所”，并选择一个地点。当屏幕切换到写入状态时，点击地点名称，并查看标签内容。记下所显示的

URL（见图4-3），因为NXP TagWriter需要这个信息写入。把手机贴近标签，应用程序就写标签了。

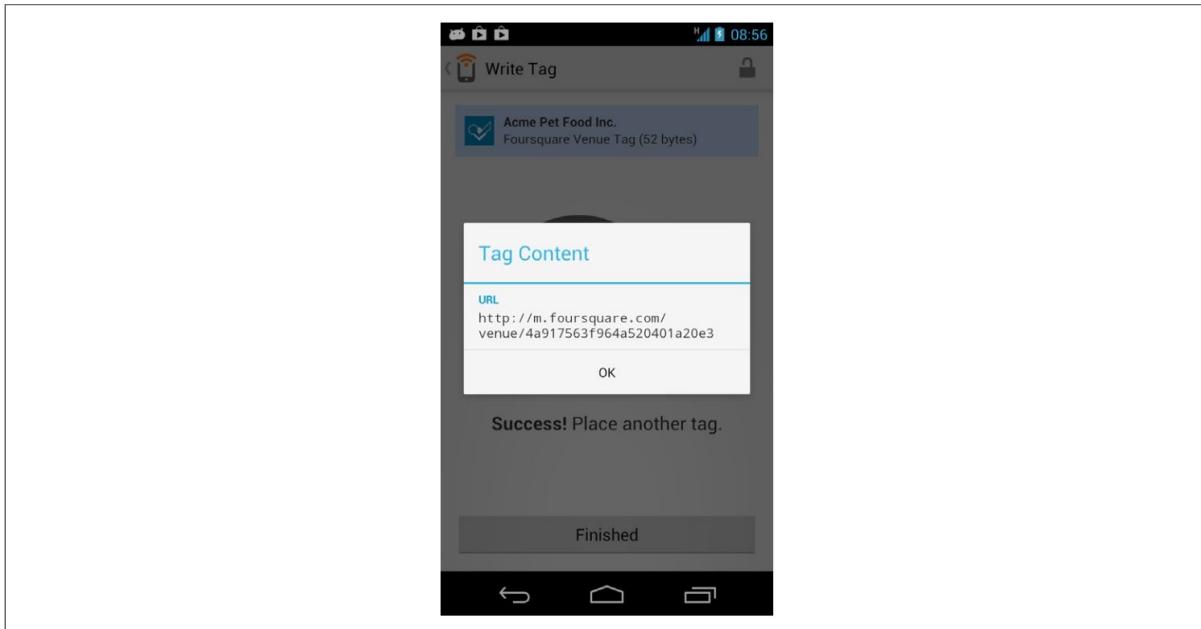


图4-3：在TagStand Writer的写标签屏幕，你可以点击地点名称来看这个地点的完整URL信息

NFC TagWriter

打开应用程序并选择“创建、写入和存储”。选择“新建”，然后从菜单中选择“URI”。在描述字段中输入“Foursquare签到”。练习时写描述是有用的，以便可以通过智能海报记录写入标签。

Foursquare签到地点如下：<http://m.foursquare.com/venue/venueid>，其中venueid是一个很长的十六进制字符串。复制一个在前面的TagStand Writer例子里用到的地点。不要忘记包含<http://>。点击“下一步”。把手机贴近标签，应用程序就写标签了。

Samsung TecTiles

确保你已经安装了Foursquare。打开TecTiles，点击“新建任务”。命名新任务，然后点击“确定”。点击“添加”创建新任务。点击“应用”。从应用程序列表中选择Foursquare。点击“写入标签”。把手机贴近标签，应用程序就写标签了。

AppLauncher NFC

打开应用程序，选择Foursquare。把手机贴近标签，应用程序就写标签了。

试一试写5个标签，完成后，关闭所有的应用程序，然后把手机贴近标签。这些标签的响应信息内容如表4-1所示。

接下来，打开NXP TagInfo。把手机贴近标签，应用程序将读标签。选择NDEF，你可以看到这几个标签的NDEF详细信息（见表4-1）。

表4-1：每个应用程序的输出信息

App	Record	TNF	Record type	Payload	Action
NFC Task Launcher	1	MIME	x/nfctl	enZ:Foursquare;c:4a917563f964a520401a20e3	Attempts to launch Task Launcher app; if successful, passes URL to Foursquare app
	2	External	android.com:pkg	com.jwsoft.nfcactionlauncher	
Tagstand Writer:	1	Well-Known	U	http://m.foursquare.com/venue/4a917563f964a520401a20e3	Launches Foursquare app, takes you to venue check-in screen
NXP TagWriter	1	Well-Known	Sp		Launches Foursquare app, takes you to venue check-in screen
	1.1		U	http://m.foursquare.com/venue/4a917563f964a520401a20e3	
	1.2		T	Foursquare check-in	
Samsung TecTiles	1	Well-Known	U	tectile://www/samsung.com/tectiles	Attempts to launch TecTiles app
	2	Well-Known	T	· enTask · · · Foursquare · com.joelapenna... ^a	
App Launcher NFC	1	External	android.com:pkg	com.joelapenna.foursquared	Launches Foursquare app only

^a三星TecTiles消息包含有不可打印的字符，实际字符不是很重
要，所以已经换成了一个中间点，行也被截断以省略号代替，你可以

通过用TecTiles写标签，再用NXP TagInfo读标签的方式看到这些标签的完整字节流信息。

正如你所见，每个应用程序的工作方式都不一样。这里有4种基本的工作方式：

- 打开Foursquare应用，然后让用户做余下的工作（App Launcher NFC）
- 发送一个URI，然后让操作系统做余下的工作（Tagstand Writer）
- 启动本应用后进而启动Foursquare应用（NFC Task Launcher，Samsung TecTiles）
- 发送一个智能海报（NXP TagWriter）

第1种方式只使用一条NDEF记录，TNF设置为“External”。记录类型就是Android应用程序，内容就是实际应用程序的名称，像这样：

```
TNF: External
Record type: android.com:pkg
com.joelapenna.foursquared
```

使用第1种方式，只需要告诉Android启动哪个应用程序。

第2种方式也只使用一条NDEF记录，TNF设置为“Well-Known”，记录类型设置为“U”（URI），内容就是实际的地址，像这样：

```
TNF: Well-Known
Record Type: U
http://m.foursquare.com/venue/4a917563f964a520401a20e3
```

使用第2种方式，需要告诉Android要打开东西的URI，由操作系统决定哪个应用程序最适合打开它。这有点像让Windows决定由哪个应用程序打开一个特定扩展名的文件。如果Foursquare不在设备上，Google Play会打开来处理这些URL。

第3种方式使用一个由两条NDEF记录组成的NDEF消息。对于NFC Task Launcher和Samsung TecTiles，原应用程序读取标签，然后启动Foursquare。NFC Task Launcher使用一条包含Foursquare的地点信息

的MIME类型记录，和一条确保该应用程序已安装的External AAR记录。TecTiles采用类似的方法。TecTiles使用一条包含自定义tectile://URL的URI记录来启动应用程序。它把Foursquare信息转换成一条文本记录。遗憾的是，TecTiles只启动应用程序，它不保存地点信息。当它们的标签被扫描到时，这两个应用程序使用intent filter来启动应用程序。NFC Task Launcher注册的MIME类型为x/nfctl，TecTiles则注册为它们定制的tectile://URI。在“Android标签分发系统”一节你将了解到更多关于intent filter的知识。

第4种方式采用智能海报记录。智能海报是一种更复杂的NDEF记录类型，其载荷实际上是另一个NDEF消息。嵌入在智能海报载荷里的消息包含两条NDEF记录，其中一条是URI记录，另一条是文本记录。由于智能海报记录有多条记录，它们可以提供有关URI的附加信息，如标题、图标或建议的处理措施。

你能看到某些应用，如TecTiles和NFC Task Launcher，写Android应用记录以启动自己的应用程序，而不是Foursquare。然后，再由它们的应用程序去启动Foursquare。这大概使它们能够跟踪应用程序的使用情况，即使最后的结果是打开另一个不同的应用程序。这更复杂，但是可以收集更多应用程序使用情况的信息。

一个标签写入应用：Foursquare签到

在本节中，你将编写自己的标签写入应用，以此来更好地理解这是如何工作的。这个应用程序非常简单，它搜索NDEF标签，如果找到，就将一个NDEF消息写进标签。

对于这个项目，你需要准备：

- 一部具有NFC功能的Android手机
- 5个RFID标签
- Android上的Foursquare应用 (<http://bit.ly/foursq-android>) 和 Foursquare (<http://www.foursquare.com>) 账户

和在第3章里的操作一样，创建一个新项目。使用Cordova创建一个项目，添加Android平台，并安装插件：

```
$ cordova create ~/FoursquareCheckin com.example.checkin FoursquareCheckin ①
$ cd ~/FoursquareCheckin ②
$ cordova platform add android
$ cordova plugin add https://github.com/chariotsolutions/phonegap-nfc
```

①Windows用户用%UserProfile%\FoursquareCheckin代替~/FoursquareCheckin。

②Windows用户用/d%UserProfile%\FoursquareCheckin代替~/FoursquareCheckin。

现在，你可以通过编辑HTML和JavaScript文件来编写应用程序了。index.html文件在刚才创建的应用程序的www目录下，index.js文件在www/js目录下。打开它们，清空文件，从头开始你自己的应用程序。编辑index.html文件如下：

```
<!DOCTYPE html>
<html>
```

```
<head>
  <title>Foursquare Check-In Tag Writer</title>
  <style>body { margin: 20px }</style>
</head>
<body>
  <p>Foursquare Check-In Tag Writer</p>
  <div class="app">
    <div id="messageDiv">No tag found</div>
  </div>
  <script type="text/JavaScript" src="cordova.js"></script>
  <script type="text/JavaScript" src="js/index.js"></script>
  <script type="text/JavaScript">
    app.initialize();
  </script>
</body>
</html>
```

把一条NDEF记录写入标签

接下来，通过写index.js文件来格式化NDEF记录，并把一个NDEF消息写入标签。现在，你只要简单地硬编码大部分参数就行。之前，你已看过通过多种方式来触发Foursquare签到。首先，只做最简单的事：用一条Android应用记录来启动Foursquare应用程序。

从写一个变量开始：

```
var app = {
    messageToWrite: [], // message to write on next NFC event
```

接下来initialize（）函数开始做事了，bindEvents（）函数建立一个事件监听器来检测设备何时准备好：

```
// Application constructor
initialize: function() {
    this.bindEvents();
    console.log("Starting Foursquare Checkin app");
},
/*
    bind any events that are required on startup to listeners:
*/
bindEvents: function() {
    document.addEventListener('deviceready', this.onDeviceReady, false);
},
```

接着清除屏幕并添加一个事件监听器来监听是否搜索到标签：

```
/*
    this runs when the device is ready for user interaction:
*/
onDeviceReady: function() {
    app.clear();

    nfc.addTagDiscoveredListener(
        app.onNfc,           // tag successfully scanned
```

```
function (status) { // listener successfully initialized
    app.makeMessage();
    app.display("Tap an NFC tag to write data");
},
function (error) { // listener fails to initialize
    app.display("NFC reader failed to initialize"
        + JSON.stringify(error));
}
),
},
},
```

NFC事件处理函数onNfc ()，进行写入标签操作，就像这样：

```
/*
    called when a NFC tag is read:
*/
onNfc: function(nfcEvent) {
    app.writeTag(app.messageToWrite);
},
```

接下来是display () 和clear () 函数：

```
/*
    appends @message to the message div:
*/
display: function(message) {
    var label = document.createTextNode(message),
        lineBreak = document.createElement("br");
    messageDiv.appendChild(lineBreak); // add a line break
    messageDiv.appendChild(label); // add the text
},
/*
    clears the message div:
*/
clear: function() {
    messageDiv.innerHTML = "";
},
```

如下的makeMessage () 和writeTag () 函数，是NFC功能插件里的两个对象定义的。这些NFC对象可访问设备的NFC读取器，NDEF对象定义了NDEF记录和信息格式。

```
makeMessage: function() {
    // Put together the pieces for the NDEF message:
    var tnf = ndef.TNF_EXTERNAL_TYPE,          // NDEF Type Name Format
        recordType = "android.com:pkg",         // NDEF Record Type
        payload = "com.joelapenna.foursquared", // content of the record
        record,                  // NDEF record object
        message = [];           // NDEF Message to pass to writeTag()

    // create the actual NDEF record:
    record = ndef.record(tnf, recordType, [], payload);
    // put the record in the message array:
    message.push(record);
    app.messageToWrite = message;
},
writeTag: function(message) {
    // write the record to the tag:
    nfc.write(
        message,          // write the record itself to the tag
        function () {    // when complete, run this callback function:
            app.display("Wrote data to tag."); // write to the message div
        },
        // this function runs if the write command fails:
        function (reason) {
            alert("There was a problem" + reason);
        }
    );
}
}; // end of app
```

你已经知道一条NDEF记录包括TNF、记录类型和载荷。NDEF对象有创建NDEF记录的功能。NDEF消息只是一个NDEF记录数组。为了创建一条新的NDEF记录，需要传入4个参数：

类型名称格式 (TNF)

一个3bit值。TNF值内置在NDEF对象常量里，所以可以使用这些常量来引用它们。

记录类型

一个字符串或0~255字节的字节数组，表示记录类型。NDEF对象内置了许多记录类型常量，尽管不是全部，但在本例中你可以直接用这些常量。

记录ID

一个字符串或0 ~ 255字节的字节数组。之前学过，记录ID是可选字段，因此你可以使用一个空数组，但是这个值不能为空。

载荷

一个字符串或者0 ~ (232-1) 字节的字节数组。同样，你可以使用一个空数组，但这个值不能为空。

在上面例子的makeMessage () 函数里，TNF等于TNF_EXTERNAL_TYPE，这是NDEF对象里的TNF常量；记录类型是android.com: pkg，意味着载荷是Android应用程序记录，没有指定记录ID，所以发送一个空数组；载荷的内容是启动的应用程序的名称，com.joelapenna.foursquared。

保存index.html和index.js文件，然后到这个新应用程序的根目录，并运行它：

```
cordova run
```

运行应用程序，然后让设备轻触标签，你应该看到标签ID（见图4-4左图）。点击链接，应用程序将写标签，写成功后给出通知（见图4-4右图）。关闭该应用程序，让设备轻触标签，它将启动Foursquare。

完整的源代码可以在GitHub (<http://bit.ly/4square-checkin>) 上找到。

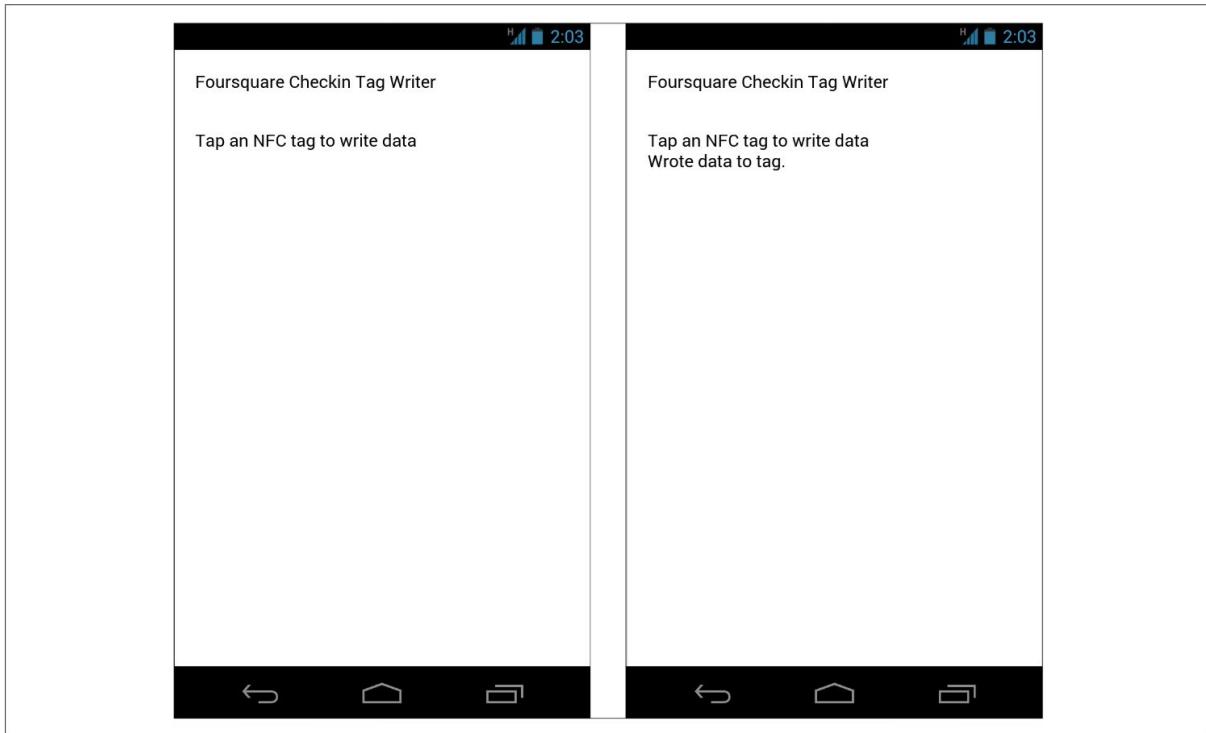


图4-4: Foursquare签到应用程序，等待一个标签（左图）和写入标签（右图）

写不同的记录类型

现在你的应用程序可以写标签，然后让这个标签来启动另一个应用程序了，但它还不能做那些你在前面的例子中所看到的功能。如果你的应用程序能模拟前面5个应用程序的功能，这将是多么伟大啊！要做到这一点，先新建一个项目“FoursquareAdvanced”。安装NFC插件，然后从之前的Foursquare签到应用程序里复制index.html和index.js文件：

```
$ cordova create ~/FoursquareAdvanced com.example.advanced FoursquareAdvanced ①
$ cd ~/FoursquareAdvanced ②
$ cordova platform add android
$ cordova plugin add https://github.com/chariotsolutions/phonegap-nfc
$ cp ~/FoursquareCheckin/www/index.html ~/FoursquareAdvanced/www/.
$ cp ~/FoursquareCheckin/www/js/index.js ~/FoursquareAdvanced/www/js/.
```

①Windows用户用%UserProfile%\FoursquareAdvanced代替~/FoursquareAdvanced。

②Windows用户用/d%UserProfile%\FoursquareAdvanced代替~/FoursquareAdvanced。

首先，在index.html文件里添加一个选项菜单，让用户选择需要模拟的应用程序。下面是新的index.html文件：

```
<!DOCTYPE html>
<html>
  <head>
    <title>Foursquare Check-In Tag Writer - Advanced</title>
    <style>
      body { margin: 20px }
    </style>
  </head>
  <body>
    <p>Foursquare Check-In Tag Writer - Advanced</p>
    <div class="app">
      <form>
```

```
Write a tag like: <br />
<select id="appPicker">
  <option value="1">NFC Task Launcher</option>
  <option value="2">TagStand Writer</option>
  <option value="3">NXP TagWriter</option>
  <option value="4">Samsung TecTiles</option>
  <option value="5">App Launcher NFC</option>
</select>
</form>
<div id="messageDiv"></div>
</div>
<script type="text/JavaScript" src="cordova.js"></script>
<script type="text/JavaScript" src="js/index.js"></script>
<script type="text/JavaScript">
  app.initialize();
</script>
</body>
</html>
```

当用户选择菜单中的某个选项时，需要有一个方便的表单，通过这个表单可以决定如何格式化标签。保存index.html文件，打开index.js文件。修改makeMessage（）函数来处理这个问题。新函数通过构建几种不同类型的NDEF消息来模拟选中的应用程序。首先，添加一个新的局部变量来读取HTML表单元素，找出用户希望模拟的应用程序。其余的局部变量和之前的相同，但它们的值会在运行时产生：

```
makeMessage: function() {
  // get the app type that the user wants to emulate from the HTML form:
  var appType = parseInt(appPicker.value, 10),
    tnf,          // NDEF Type Name Format
    recordType,   // NDEF Record Type
    payload,      // content of the record
    record,       // NDEF record object
    message = [];// NDEF Message to pass to writeTag()
```

makeMessage（）函数的其余部分将被彻底改变。局部变量定义之后，根据所选模拟应用程序的不同来格式化不同的NDEF消息。由于从HTML表单选项菜单返回的是一个整数，你可以使用一个switch-case语句来切换分支。如你所见，每个case分支写一种应用程序格式的记录。Case 1创建一条MIME类型的记录，其内包含启动应用程序的指令，然后一条Android应用程序记录告诉Android系统启动哪个应用程序：

```
switch (appType) {
    case 1: // like NFC Task Launcher
        // format the MIME media record:
        recordType = "x/nfctl";
        payload = "enZ:Foursquare;c:4a917563f964a520401a20e3";
        record = ndef.mimeMediaRecord(recordType, payload);
        message.push(record); // push the record onto the message

        // format the Android Application Record:
        tnf = ndef.TNF_EXTERNAL_TYPE;
        recordType = "android.com:pkg";
        payload = "com.jwsoft.nfcactionlauncher";
        record = ndef.record(tnf, recordType, [], payload);
        message.push(record); // push the record onto the message
        break;
```

Case 2创建一条Well-Known类型的记录，RTD设为URI。因为URI有标准格式，NDEF规范使用URI识别码来代替标准URI头，以此缩短载荷。例如，表4-2显示了前几个URI识别码。

表4-2：一些URI识别码

URI 识别码 (UIC)	含义
0x00	无，用于用户自定义头
0x01	<i>http://www.</i>
0x02	<i>https://www.</i>
0x03	<i>http://</i>

像ftp://、mailto: 和file:///等常用的URI头都有自己的UIC。有关UIC的完整列表，请参阅URI记录类型定义文档，在NFC论坛网站的NFC规范内有这些内容。本书附录A中也有。

为了使用UIC，你需要将URI字符串转换为字节数组，然后把UIC放在数组的开始处。在Case 2中，添加0x03，代替http://：

```
case 2: // like Tagstand Writer
    // format the URI record as a Well-Known type:
    tnf = ndef.TNF_WELL_KNOWN;
    recordType = ndef.RTD_URI; // add the URI record type
    // convert to an array of bytes:
    payload = nfc.stringToBytes(
        "m.foursquare.com/venue/4a917563f964a520401a20e3");
    // add the URI identifier code for "http://":
    payload.unshift(0x03);
    record = ndef.record(tnf, recordType, [], payload);
    message.push(record); // push the record onto the message
    break;
```

Case 3是特殊的例子，因为它创建了智能海报消息。智能海报是一种特殊的Well-Known类型，它们的记录里包含NDEF消息。在下一个例子中你将看到，所要构造的智能海报记录的载荷是一个记录数组。该数组就是消息。

在第5章中，你将看到如何通过递归的方式将智能海报记录内容作为消息来提取：

```
case 3: // like NXP TagWriter
    // The payload of a Smart Poster record is an NDEF message
    // so create an array of two records like so:
    var smartPosterPayload = [
        ndef.uriRecord(
            "http://m.foursquare.com/venue/4a917563f964a520401a20e3"),
        ndef.textRecord("foursquare checkin"),
    ];
    // Create the Smart Poster record from the array:
    record = ndef.smartPoster(smartPosterPayload);
    // push the Smart Poster record onto the message:
    message.push(record);
    break;
```

Case 4构建了一条URI记录和一条内含一些二进制数据的文本记录。正如你在Case 2中看到的，你需要将URI转换为字节数组，然后将UIC插到数组前端。在这里，由于Samsung使用了自定义URI头（tectile: //），所以UIC为0x00，URI原封不动地写入。TecTiles写的第二条记录TNF为01（Well-Known），类型是“T”。

在载荷中，Samsung还使用了一个专有令牌。当使用NXP的TagReader读由TecTiles写的标签时就能确定这一点。下面是按照Samsung应用程序所需的消息格式复制的代码：

```
case 4: // like TecTiles
    // format the record as a Well-Known type
    tnf = ndef.TNF_WELL_KNOWN;
    recordType = ndef.RTD_URI; // add the URI record type
    var uri = "tectiles://www.samsung.com/tectiles";
    payload = nfc.stringToBytes(uri);
    var id = nfc.stringToBytes("0");
    // URI identifier 0x00 because there's no ID for "tectile://":
    payload.unshift(0x00);
    record = ndef.record(tnf, recordType, id, payload);
    message.push(record); // push the record onto the message
    // text record with binary data
    tnf = ndef.TNF_WELL_KNOWN;
    recordType = ndef.RTD_TEXT;
    payload = [];
    // language code length
    payload.push(2);
    // language code
    payload.push.apply(payload, nfc.stringToBytes("en"));
    // Task Name
    payload.push.apply(payload, nfc.stringToBytes("Task"));
    // 4-byte token proprietary to TecTiles:
    payload.push.apply(payload, [10, 31, 29, 19]);
    // Application Name
    payload.push.apply(payload, nfc.stringToBytes("Foursquare"));
    // NULL terminator
    payload.push(0);
    // Activity to launch
    payload.push.apply(payload, nfc.stringToBytes(
        "com.joelapenna.foursquared.MainActivity"));
    // NULL terminator
    payload.push(0);
    // Application packageName
    payload.push.apply(payload, nfc.stringToBytes(
        "com.joelapenna.foursquared"));
    id = nfc.stringToBytes("1");
    record = ndef.record(tnf, recordType, id, payload);
    message.push(record); // push the record onto the message
    break;
```



对于某些标签来说，TecTiles记录太大了而无法写入，如Mifare Ultralight，容量很小。

Case 5构造了一条可直接打开Foursquare的Android应用类型记录。就像之前已经做过的Android应用程序记录。TNF为“External”，记录类型为android.com: pkg，应用程序名称就是载荷：

```
case 5: // like App Launcher NFC
    // format the Android Application Record:
    tnf = Ndef.TNF_EXTERNAL_TYPE;
    recordType = "android.com:pkg";
    payload = "com.joelapenna.foursquared";
    record = Ndef.record(tnf, recordType, [], payload);
    message.push(record); // push the record onto the message
    break;
} // end of switch-case statement
```

最后，在makeMessage（）函数末尾，设置要写入的消息并通知用户轻触标签：

```
app.messageToWrite = message;
app.display("Tap an NFC tag to write data");
}, // end of makeMessage()
```

保存index.js文件，运行应用程序（确认你的应用程序里还有writeTag（）函数）：

```
$ cordova run
```

当你启动应用程序时，会得到一个下拉菜单，可以让你模拟在表4-1中看到的任何应用程序。从菜单中选择一个应用程序，然后用手机轻触标签。图4-5显示了应用程序的界面。完整的源代码可以在GitHub (<http://bit.ly/4squareadv>) 上找到。

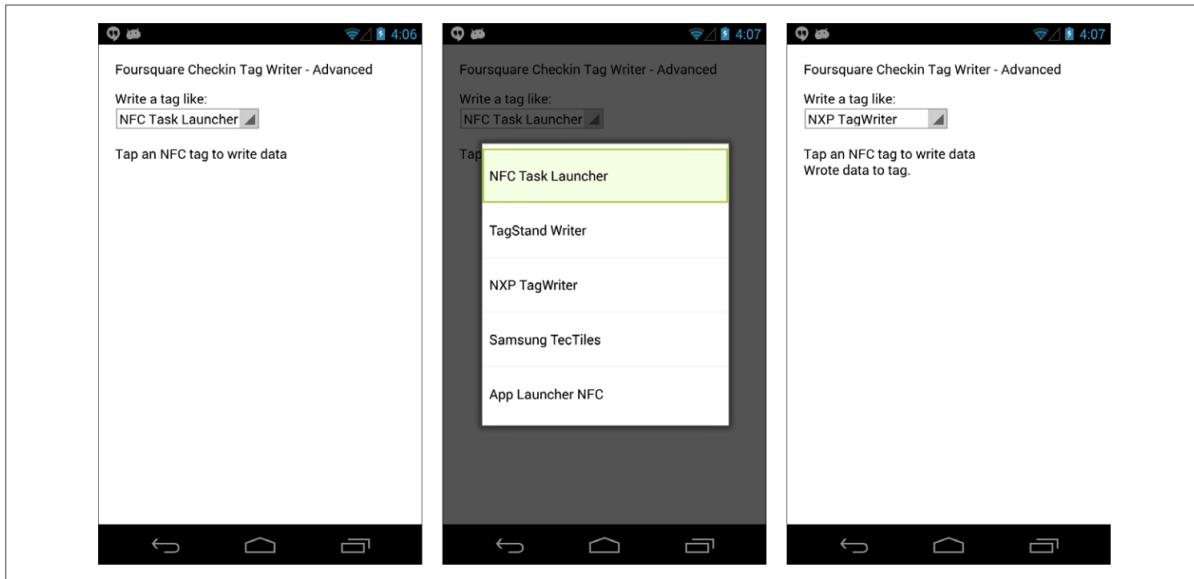


图4-5：Foursquare签到高级应用程序；初始界面（左图），显示选项菜单（中图），并写入标签（右图）

PhoneGap-NFC的NDEF辅助函数概述

如你所见，上节例子的应用程序所做的模拟功能在PhoneGap-NFC库里也有。PhoneGap-NFC库的NDEF对象有一些辅助函数，这些函数可用来写NDEF记录。主要使用的一个函数是`ndef.record()`，它需要你传给它TNF、记录类型、ID和有效载荷：

```
// Caller specifies the TNF and record type
record = ndef.record(tnf, recordType, id, payload);
// example:
record = ndef.record(ndef.TNF_EXTERNAL_TYPE, "android.com:pkg",
[], "com.joelapenna.foursquared");
```

还有一个MIME记录辅助函数，只需要传入MIME类型和载荷：

```
// TNF: MIME media (02)
record = ndef.mimeMediaRecord(mimeType, payload);
// example:
record = ndef.mimeMediaRecord("text/json", '{"answer": 42}');
```

URI辅助函数构建了一条TNF 01（Well-Known）、记录类型是“U”的记录，代表发送的是URI（就像你在之前的例子中看到的）。辅助函数将会用UIC来缩短URI。在本例中，`m.foursquare.com`前的`http://`将被`0x03`替换并写入标签：

```
// TNF: Well-Known Type(01), RTD: URI ("U")
record = ndef.uriRecord(uri);
// example:
record = ndef.uriRecord("http://m.foursquare.com/");
```

文本记录辅助函数使用Well-Known TNF和文本记录类型来创建文本记录。如果没有指定语言，则默认为英语：

```
// TNF: Well-Known Type(01), RTD: Text ("T")
record = ndef.textRecord(text, language);
```

```
// example:  
// defaults to English, since no language specified:  
record = ndef.textRecord("How are you doing?");
```

智能海报辅助函数从其他记录构建智能海报：

```
//TNF: Well-Known Type (01), RTD: Smart Poster ("Sp")  
record = ndef.smartPoster(ndefMessage);  
// example:  
record = ndef.smartPoster (  
    // URI record:  
    ndef.uriRecord("http://m.foursquare.com/venue/4a917563f964a520401a20e3"),  
    // text record:  
    ndef.textRecord("foursquare checkin")  
);
```

还有一个辅助函数你在本章没见过，就是空记录辅助函数。它创建一个空记录，让你来填充：

```
// TNF: Empty (00)  
record = ndef.emptyRecord();  
// example:  
record = ndef.emptyRecord(); // it's empty!
```

有了这些辅助函数，你可以构造本书所涉及的所有类型的NDEF消息。

总结

我们将在后面的章节中更进一步深入讲解NDEF记录和消息。本章介绍的关键概念如下：NFC格式的标签包含NDEF消息。NDEF消息是由一条或多条NDEF记录构成的。请先别急着讨论标签、消息和记录之间如何交互，在本书的后面你还会学到设备与设备之间直接的消息交换，而无须通过标签。

所有的NDEF记录都有一种类型。在规范里，它有时被称为载荷类型，有时也称为记录类型，或者只简单地称为类型。类型名称格式（TNF）把类型分为几种，并解释了这几种类型：

TNF 01 (Well-Known)

由NFC论坛记录类型定义（RTD）规范描述记录类型。

TNF 02 (MIME)

由MIME RFC（RFC 2046）规范描述记录类型。

TNF 03 (URI)

由URI RFC（RFC 3986）规范描述如何生成一个有效的类型。

TNF 04 (External)

NFC论坛RTD规范描述如何生成你自己的类型。

理论上NDEF消息有8种TNF，但大部分应用程序只用到3种：Well-Known、MIME和External。Well-Known的TNF类型里包括大量有用的记录类型，包括文本类型、URI、智能海报，以及点对点交换时用到的各种载体类型。在第8章你将学到更多关于点对点消息的内容。在本书中你会看到文本消息和URI类型的记录。MIME的TNF涵盖了所

有的互联网媒体类型，你也可以用它来制作自定义类型，就像你在第6章中所看到的。External的TNF多用于像Android应用程序记录这样的记录类型。

本书中你不会看到很多智能海报类型的例子。就像你在“写不同的记录类型”一节看到的智能海报例子，智能海报记录的载荷本身就是一个NDEF消息。所以为了读取智能海报，你首先必须解析它的外围消息，然后再解析它包含的载荷信息。我们觉得对于大多数应用程序来说这是多余的，更好的做法是简单地使用多条记录的NDEF消息。

有多种方式来实现与NDEF消息同样的目的。如何接收消息取决于正在接收它的设备的操作系统。在下一章中，你将看到Android系统为你的应用程序提供了一些方法来过滤不同类型的NDEF消息。

第5章 监听NDEF数据

每一个设计优秀的应用程序都依赖于对相应事件的良好监听，不管是用户输入事件、网络事件还是其他形式的事件。NFC应用程序也一样，NFC程序也总是在监听着NDEF信息，解析和过滤信息的类型和内容，并相应地进行响应。在第4章中，你已经知道如何使用PhoneGap-NFC插件编写NDEF信息，但是没有深入到监听的部分。

监听NDEF信息有两种主要的方式：一种是你可以直接编码，当应用程序运行在前台时直接监听；另一种是让设备的操作系统去监听，然后当它遇到与应用程序相关的信息时再通知应用程序。PhoneGap-NFC插件提供了几种NFC事件让应用程序可以监听，Android也提供了标签分发系统，它能让你通知操作系统什么样的标签是你感兴趣的。NFC应用程序可以通过在Android Manifest中编写intent filter来充分利用标签分发系统，它会告诉操作系统什么样的标签是需要经过应用程序的。当然，你也可以通过使用前台分发系统（foreground dispatch system）覆盖其他应用程序的过滤器来明确地监听部分或者全部标签。在本章中，你会学到如何使用PhoneGap-NFC的事件监听器和Android的标签分发系统来监听和过滤NDEF信息。

PhoneGap-NFC的事件监听

到现在为止，你还没有指定Android应该如何处理标签，而且之前也只有一个事件监听器tagDiscoveredListener。这个事件监听器使用前台分发系统告诉Android你想让应用程序接收所有NFC相关的信息，而不是让这些信息转发到其他的应用程序。如果你想让这些信息都经过你的应用程序，用这种方式可以处理得很好，当然你也可能看到一些其他的应用程序，比如TagWriter、TagInfo，它们也同样是使用前台分发系统这种方式。当你的应用程序处于前台时，它能优先得到所有的事件以及NFC等的通知。这样就意味着你能决定想监听什么以及想忽略什么。

PhoneGap-NFC插件提供了4种NFC相关事件监听器：

发现标签监听器

监听所有与读取器硬件能兼容的标签，这是最常用的监听器。

NDEF格式监听器

监听可以格式化接收NDEF信息的所有兼容标签。

NDEF监听器

监听任何包含NDEF信息的标签，如果监听到合法的NDEF信息，该监听器将产生一个事件。

MIME类型监听器

这种类型的监听器仅仅监听包含MIME类型的NDEF信息，这是所有事件监听器中最特别也是最有用的。你可以使用MIME类型的监听器过滤包含有指定MIME类型的信息，这样它就能忽略其他的MIME类型，同样也可以使用它过滤掉那些不包含任何类型的

信息。然而，你无法为不同的类型设置两个MIME类型的监听器。

NDEF中的MIME媒体类型 (Media-Type)

“类型”这个词在谈到NFC时有很多意思，如果你还是感觉到困惑，请重新看一下“TNF（类型名称格式）”以及“载荷类型”章节，那里有详细的说明。本章主要关注MIME媒体类型。MIME（起初是Multipurpose Internet Mail Extensions，多用途互联网邮件扩展）是一个互联网标准，支持各种非文本媒体类型。对于MIME类型的完整列表，请参阅互联网编号分配机构（Internet Assigned Numbers Authority, IANA）关于MIME类型的文档（<http://bit.ly/media-types>）。NFC论坛规范编写者意识到，他们没有必要再重新发明一种媒体类型的协议，因为MIME已经做得很好了。

对于包含TNF MIME媒体类型的NDEF记录来说，通常都是使用预先存在的类似于text/plain的MIME类型作为记录类型来描述有效载荷的内容。NDEF规范中写着“使用非注册的媒体类型是不被允许的”，但是我们还是做了，为什么呢？因为MIME类型将你的NFC标签与其他标签区别开来是非常简单和方便的。比如，在第6章中使用的Hue灯，我们定义了一种text/hue的MIME类型，在“NDEF实战”一节中你也能看到NFC Task Launcher使用了一种叫作x/nfctl的MIME类型。它们都不是MIME规范中常见的类型。

Android提供了intent filter（详见本章后面的“Android标签分发系统”一节），让应用程序可以注册监听它关注的NFC标签。当系统遇到应用程序注册监听的NFC标签时，它将启动你的应用程序并将数据也包含进来。NFC Task Launcher就注册了MIME类型x/nfctl的监听，在第6章中你也会做相同的事情。

你可以为同一个应用程序注册多个监听器，即使指定的标签满足多个监听，但仍然只会执行最符合要求的那一个监听。例如，一个plain-text类型的信息标签，它可能触发所有的这些监听器。它是可读

的，可以格式化成NDEF，是NDEF格式的，是合法的MIME类型（text/plain）。然而，一旦MIME类型是最合适的监听器，那么这个事件监听就将被执行。

你也可以使用多个监听器叠加在一起。比如，如果你知道你的标签大部分都是plain-text类型标签，那么可以指定一个MIME类型的监听器捕捉它们，然后使用一个NDEF类型的监听器捕捉其他类型的NFC标签，以及使用NDEF格式监听器搜索空标签并提示用户填写些什么。

在下一个程序中，我们将用行动来证明这一点。在这个应用中，你将实现4个监听器，并且使用不同的标签去触发每一个。

一个NDEF读取器应用

对于这个项目，你需要：

- 具有NFC功能的Android手机。
- 至少4个NFC标签（为了达到更好的效果，最好是使用NFC论坛上推荐的，详见“设备与标签类型匹配”一节获取更多设备和标签的信息）。

我们同样也知道你已经通过第3章安装好了本章例子所需要的软件。

首先使用cordova create命令创建一个新的项目：

```
$ cordova create ~/NdefReader com.example.ndefreader NdefReader❶
$ cd ~/NdefReader❷
$ cordova platform add android
$ cordova plugin add https://github.com/chariotsolutions/phonegap-nfc
```

❶Windows用户应该输入%UserProfile%\NdefReader替代~/NdefReader。

❷Windows用户应该输入/d%UserProfile%\NdefReader替代~/NdefReader。

你也可以从NFC读取器应用中复制index.html和index.js文件作为这个项目的基础：

```
$ cp ~/NfcReader/www/index.html ~/NdefReader/www/.
$ cp ~/NfcReader/www/js/index.js ~/NdefReader/www/js/.
```

本项目的index.html应该是这样的：

```
<!DOCTYPE html>

<html>
  <head>
    <title>NDEF Reader</title>
  </head>
  <body>
    <div class="app">
      <p>NDEF Reader</p>
      <div id="messageDiv"></div>
    </div>
    <script type="text/JavaScript" src="cordova.js"></script>
    <script type="text/JavaScript" src="js/index.js"></script>
    <script type="text/JavaScript" >
      app.initialize();
    </script>
  </body>
</html>
```

监听多个事件

index.js文件跟NFC读取器应用中的文件一样拥有相同的基本信息。initialize () 和bindEvents () 函数以及display () 和clear () 函数都将保留，onDeviceReady () 函数也一样启动，只是在addTagDiscooveredListener后面添加3个新的监听器。新的函数代码如下：

```
/*
  this runs when the device is ready for user interaction:
*/
onDeviceReady: function() {

  nfc.addTagDiscoveredListener(
    app.onNonNdef,          // tag successfully scanned
    function (status) {      // listener successfully initialized
      app.display("Listening for NFC tags.");
    },
    function (error) {        // listener fails to initialize
      app.display("NFC reader failed to initialize "
        + JSON.stringify(error));
    }
  );

  nfc.addNdefFormattableListener(
    app.onNonNdef,          // tag successfully scanned
    function (status) {      // listener successfully initialized
      app.display("Listening for NDEF Formattable tags.");
    },
    function (error) {        // listener fails to initialize
      app.display("NFC reader failed to initialize"
        + JSON.stringify(error));
    }
  );

  nfc.addNdefListener(
    app.onNfc,              // tag successfully scanned
    function (status) {      // listener successfully initialized
      app.display("Listening for NDEF messages.");
    },
  );
}
```

```
        function (error) {    // listener fails to initialize
            app.display("NFC reader failed to initialize "
                + JSON.stringify(error));
        }
    );

    nfc.addMimeTypeListener(
        "text/plain",
        app.onNfc,          // tag successfully scanned
        function (status) {  // listener successfully initialized
            app.display("Listening for plain text MIME Types.");
        },
        function (error) {   // listener fails to initialize
            app.display("NFC reader failed to initialize "
                + JSON.stringify(error));
        }
    );

    app.display("Tap a tag to read data.");
},
```

正如你所见，这些监听器的结构都是一样的，唯一不同的是它们注册监听成功时显示的信息，以及它们执行的回调函数。这些都可能调用onNfc（）函数返回一个NDEF信息，而不会调用onNonNdef（）函数。

当你读取的标签不以任何形式兼容NDEF时，监听器就不会返回这个标签。比如，你可以读取13.56MHz RFID标签（它甚至不是Mifare标签：PhilipsICODE或者Texas Instruments Tag-it），你的应用除了它的UID什么也读不到。如果你使用类似于Nexus 4的设备，也一样读不了Mifare Classic标签，除了能读到UID之外，其他标签数据都无法获取。你需要单独去处理这些不同的标签。onNonNdef（）函数就是做这件事的。

新的onNfc（）函数将会调用clear（）函数来清除div中的信息，然后显示调用的监听类型和标签的详细信息。onNonNdef（）函数将会读取标签的UID以及标签的信息，然后添加一个showTag（）函数去显示从onNfc（）函数中得到的标签信息。

```
/*
Process NDEF tag data from the nfcEvent
```

```

*/
onNfc: function(nfcEvent) {
    app.clear();          // clear the message div
    // display the event type:
    app.display("Event Type:" + nfcEvent.type);
    app.showTag(nfcEvent.tag); // display the tag details
},

/*
Process non-NDEF tag data from the nfcEvent
This includes
* Non NDEF NFC Tags
* NDEF-Formatable Tags
* Mifare Classic Tags on Nexus 4, Samsung S4
  (because Broadcom doesn't support Mifare Classic)
*/
onNonNdef: function(nfcEvent) {
    app.clear();          // clear the message div
    // display the event type:
    app.display("Event Type:" + nfcEvent.type);
    var tag = nfcEvent.tag;
    app.display("Tag ID:" + nfc.bytesToHexString(tag.id));
    app.display("Tech Types:");
    for (var i = 0; i < tag.techTypes.length; i++) {
        app.display("*" + tag.techTypes[i]);
    }
},
/*
writes @tag to the message div:
*/
showTag: function(tag) {
    // display the tag properties:
    app.display("Tag ID:" + nfc.bytesToHexString(tag.id));
    app.display("Tag Type:" + tag.type);
    app.display("Max Size:" + tag.maxSize + "bytes");
    app.display("Is Writable:" + tag.isWritable);
    app.display("Can Make Read Only:" + tag.canMakeReadOnly);
},

```

做完这些后就能运行这个程序了，保存好这些文件并像之前那样运行它：

\$ cordova run

使用这个程序与你在第4章中做的Foursquare签到标签一起测试一下，然后尝试用NXP TagWriter做一个新的带有plain-text数据的标签。点击“创建（create）、写入（write）和保存（store）”，选择“新建（New）”，然后选择“Plain Text”，再输入一个文本信息。最后，点击“下一步（Next）”，然后将新标签移到设备的背面写入。之后重新打开你刚编写的NDEF读取器，读取这个标签。它应该能读取到一个ndef-mime事件。到目前为止，图5-1显示了从这个程序得到的一些不同结果。

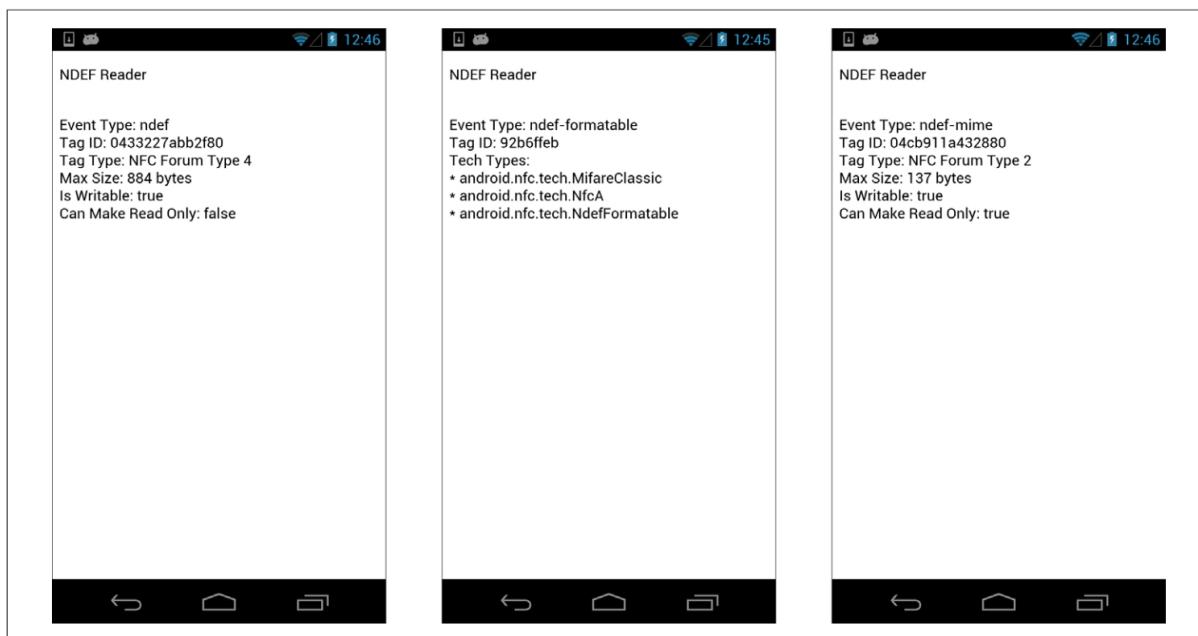


图5-1：从NDEF读取器应用程序读取的结果：一个通用的NDEF格式标签；一个空白，但可格式化NDEF标签；一个MIME类型“text/plain”的文本标签

你可能得不到任何类型“标签”的新事件，因为它被其他的事件监听器覆盖了。这样直接跨过一个与NFC读取器兼容的标签而不读取的情况是非常少见的，但至少它不是可格式化的NDEF。某些非Mifare RFID标签就会出现这样的情况（包括Texas Instrument Tag-it HF和Don的某一张信用卡），但不多。如果你碰巧为BlackBerry 7编写了PhoneGap-NFC应用，你会发现当一个给定标签进入范围内时，所有符合条件的监听器都会启动，所以，如果想让BlackBerry用户得到与Android用户同样的效果，你就得适配你的代码以达到跨平台。更多的关于跨平台的差异，请看PhoneGap-NFC插件的README文件（<http://bit.ly/nfc-readme>）。

读取NDEF消息

现在你已经得到一个可以读取任何兼容性标签的应用程序了，你可能想将其扩展成为每次读取标签时都能给出详细的NDEF消息。记住，NDEF消息是由NDEF记录组成的，当记录是智能海报（Smart Poster）时，记录的内容就是它本身这个NDEF消息。为了处理这种情况，你会需要一个showMessage（）函数，它会被showTag（）函数调用，以及一个被showMessage（）调用的showRecord（）函数。如果这条记录是智能海报，你得重新调用showMessage（）。图5-2显示了整个程序的流程。

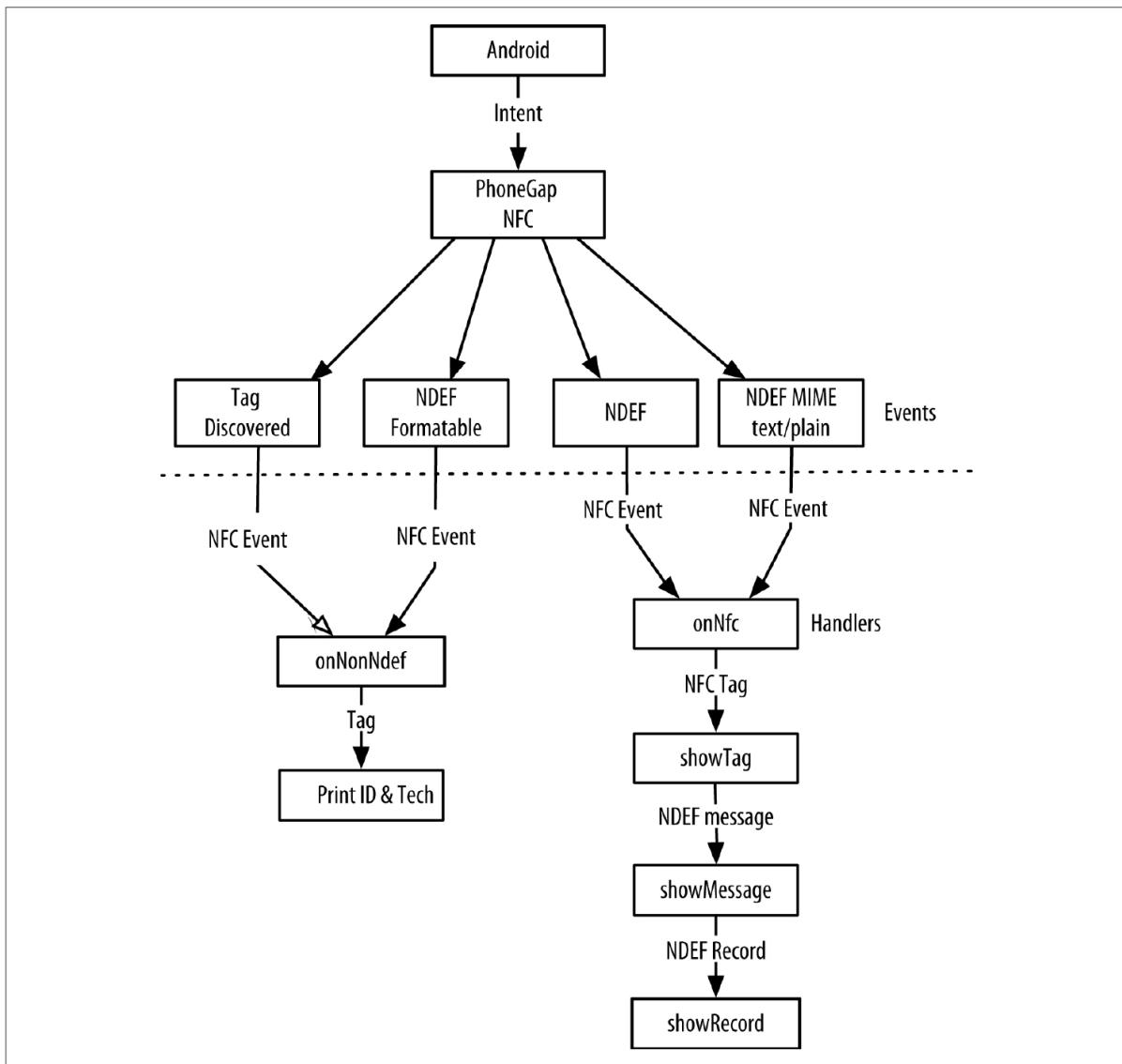


图5-2：NDEF读取器应用程序的流程

首先将一个if语句添加到showTag（）函数，这个函数会将标签上的NDEF消息发送到showMessage（）函数。

```

showTag: function(tag) {
    // display the tag properties:
    app.display("Tag ID: " + nfc.bytesToHexString(tag.id));
    app.display("Tag Type: " + tag.type);
    app.display("Max Size: " + tag.maxSize + "bytes");
    app.display("Is Writable: " + tag.isWritable);
    app.display("Can Make Read Only: " + tag.canMakeReadOnly);
  
```

```

// if there is an NDEF message on the tag, display it:
var thisMessage = tag.ndefMessage;
if (thisMessage !== null) {
    // get and display the NDEF record count:
    app.display("Tag has NDEF message with " + thisMessage.length
        + "record" + (thisMessage.length === 1 ? ".:" : "s."));

    app.display("Message Contents:");
    app.showMessage(thisMessage);
}
},

```

接下来，添加一个`showMessage ()` 函数来显示消息，这个消息是一个记录数组，所以使用第二个函数`showRecord ()` 来循环显示每一条记录。

```

/*
 iterates over the records in an NDEF message to display them:
*/
showMessage: function(message) {
    for (var thisRecord in message) {
        // get the next record in the message array:
        var record = message[thisRecord];
        app.showRecord(record);      // show it
    }
},
/*
 writes @record to the message div:
*/
showRecord: function(record) {
    // display the TNF, Type, and ID:
    app.display(" ");
    app.display("TNF:" + record.tnf);
    app.display("Type:" + nfc.bytesToString(record.type));
    app.display("ID:" + nfc.bytesToString(record.id));

    // if the payload is a Smart Poster, it's an NDEF message.
    // read it and display it (recursion is your friend here):
    if (nfc.bytesToString(record.type) === "Sp") {
        var ndefMessage = ndef.decodeMessage(record.payload);
        app.showMessage(ndefMessage);
    }

    // if the payload's not a Smart Poster, display it:
} else {
    app.display("Payload:" + nfc.bytesToString(record.payload));
}

```

```
};  
}; // end of app
```

正如你所看到的，`showMessage()` 和 `showRecord()` 相互递归调用，直到标签的所有内容都显示出来。

保存这些修改并重新运行这个程序，它应该就能读取所有兼容的标签并将标签上的任何NDEF信息提供给你了。除了显示多种NDEF事件监听器，它同时也是一个对于一般标签读取来说很好的应用。它并没有非常好地显示这些记录，因为它非常的基本，它仅仅提供了一些代码去提取信息。一个更加全面发展的应用，你应该在index.html页面中添加更多的元素来显示每条记录的信息，这样你就能缩进这些记录以及做其他能美化这个应用的事情了。

图5-3展示了从NDEF读取器应用程序读取的一些例子，你可以在GitHub (<http://bit.ly/ndef-master>) 上找到这些例子。

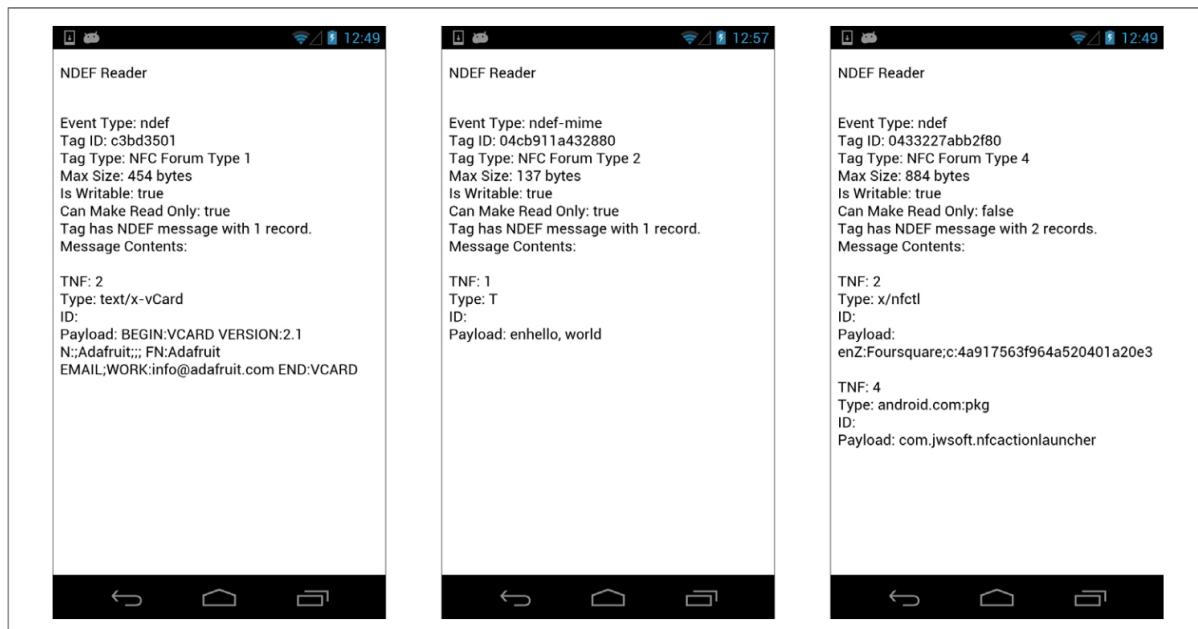


图5-3：从NDEF读取器应用程序读取的结果：一个电子名片标签（左图），一个纯文本消息（中图），一个有多条记录的标签（右图）

使用记录类型过滤标签

如果你正在做一个需要读取多种不同类型信息的应用，那么有很多种方法可以做到。你可以使用TNF来分离它们，但是在大多数情况下不会告诉你信息的作用。一般情况下，NFC论坛记录类型定义（NFC Forum Record Type Definition）会告诉你更多信息。你会看到几个常用的UFC RTD：U表示URI，T表示文本记录，Sp表示智能海报（Smart Poster）。有一些是你已经见过的，例如： android.com： pkg 作为Android应用程序的记录。当然，你也可以用任何网络标准MIME类型作为记录类型，最常见的就是text/plain。你甚至还可以制作自己的类型。接下来是NDEF读取器应用程序的showRecord（）函数，它能通过类型过滤记录。完整的源代码可以在GitHub (<http://bit.ly/ndef-master>) 上找到。

```
showTag: function(tag) {
    // display the tag properties:
    app.display("Tag ID: " + nfc.bytesToHexString(tag.id));
    app.display("Tag Type: " + tag.type);
    app.display("Max Size: " + tag.maxSize + " bytes");
    app.display("Is Writable: " + tag.isWritable);
    app.display("Can Make Read Only: " + tag.canMakeReadOnly);

    // if there is an NDEF message on the tag, display it:
    var thisMessage = tag.ndefMessage;
    if (thisMessage !== null) {
        // get and display the NDEF record count:
        app.display("Tag has NDEF message with " + thisMessage.length
            + " record" + (thisMessage.length === 1 ? "." : "s."));

        var type = nfc.bytesToString(thisMessage[0].type);
        switch (type) {
            case nfc.bytesToString(ndef.RTD_TEXT):
                app.display("Looks like a text record to me.");
                break;
            case nfc.bytesToString(ndef.RTD_URI):
                app.display("That's a URI right there");
                break;
            case nfc.bytesToString(ndef.RTD_SMART_POSTER):
```

```
        app.display("Golly! That's a smart poster.");
        break;
    // add any custom types here,
    // such as MIME types or external types:
    case 'android.com:pkg':
        app.display("You've got yourself an AAR there.");
        break;
    default:
        app.display("I don't know what " +
            type +
            " is, must be a custom type");
        break;
    }
    app.display("Message Contents: ");
    app.showMessage(thisMessage);
}
},
```

使用MIME类型过滤

因为MIME类型的记录非常有用，所以它们有自己的类型名称格式（Type Name Format），在NFC插件中也有自己的监听函数。通过MIME类型来过滤能让应用程序更快地只关注那些你关心的标签，忽略那些你不关心的标签。其他的信息则由操作系统的标签分发系统处理。作为示范，这里写了一个应用程序告诉你如何读取文本信息，而忽略其他信息。这个应用程序使用MimeTypeListener来处理。

首先使用cordova create命令创建一个新项目：

```
$ cordova create ~/MimeReader com.example.mimereader MimeReader❶
$ cd ~/MimeReader ❷
$ cordova platform add android
$ cordova plugin add https://github.com/chariotsolutions/phonegap-nfc
```

❶Windows用户应该输入%UserProfile%\MimeReader替代~/MimeReader。

❷Windows用户应该输入/d%UserProfile%\MimeReader替代~/MimeReader。

你也可以从NFC读取器应用程序中复制index.html和index.js文件作为这个项目的基础：

```
$ cp ~/NfcReader/www/index.html ~/MimeReader/www/
$ cp ~/NfcReader/www/js/index.js ~/MimeReader/www/js/.
```

index.html页面大多是相同的（唯一的变化是页面的展现）：

```
<!DOCTYPE html>
<html>
<head>
<title>NFC MIME Reader</title>
```

```
</head>
<body>
<div class="app">
<p>This app reads plain-text NFC tags.</p>
Try tags with
<ul>
<li>TNF 03 with type "text/plain"</li>
<li>TNF 01 with RTD "T"</li>
</ul>
<div id="messageDiv"></div>
</div>
<script type="text/JavaScript" src="cordova.js"></script>
<script type="text/JavaScript" src="js/index.js"></script>
<script type="text/JavaScript">
    app.initialize();
</script>
</body>
</html>
```

在index.js文件中需要改变三个地方来做这个应用程序。首先，onDeviceReady（）函数将得到一个监听函数，具体如下：

```
onDeviceReady: function() {
    nfc.addMimeTypeListener(
        "text/plain",           // listen for plain-text messages
        app.onNfc,              // tag successfully scanned
        function (status) {      // listener successfully initialized
            app.display("Tap an NFC tag to begin");
        },
        function (error) {       // listener fails to initialize
            app.display("NFC reader failed to initialize " +
                JSON.stringify(error));
        }
    );
},
```

其次，需要更改onNfc（）函数，以便它能读取标签的信息和处理里面的具体记录信息。因为你只监听MIME媒体类型记录，知道哪些信息中的记录是文本记录，所以检查是非常简单的。你需要从一开始通过语言编码来获取真正的内容，在检查有效载荷的第一字节的条件语句中，你可以看到是如何做到这一点的。

```
/*
  displays tag from @nfcEvent in message div:
*/
onNfc: function(nfcEvent) {
  var tag = nfcEvent.tag,
    text = "",
    payload;

  app.clear();
  app.display("Read tag: " + nfc.bytesToHexString(tag.id));

  // get the payload from the first message
  payload = tag.ndefMessage[0].payload;

  if (payload[0] < 5) {
    // payload begins with a small integer, it's encoded text
    var languageCodeLength = payload[0];

    // chop off the language code and convert to string
    text = nfc.bytesToString(payload.slice(languageCodeLength + 1));

  } else {
    // assume it's text without language info
    text = nfc.bytesToString(payload);
  }

  app.display("Message: " + text);
},
```

最后，删除onNonNdef（）函数。

用一些你已经格式化了的标签试试这个程序。plain-text对应的那个标签信息应该会显示在这个应用上，其他的则会由关注这些信息的应用处理。大部分标签都格式化成Foursquare，它们会自动打开Foursquare应用。你可以看到在action中使用MIME类型过滤的优势：你的应用程序将会只根据最合适的MIME类型来执行对应的事件。图5-4展示的就是这个应用的结果。完整的源代码可以在GitHub (<http://bit.ly/mreader-master>) 上找到。

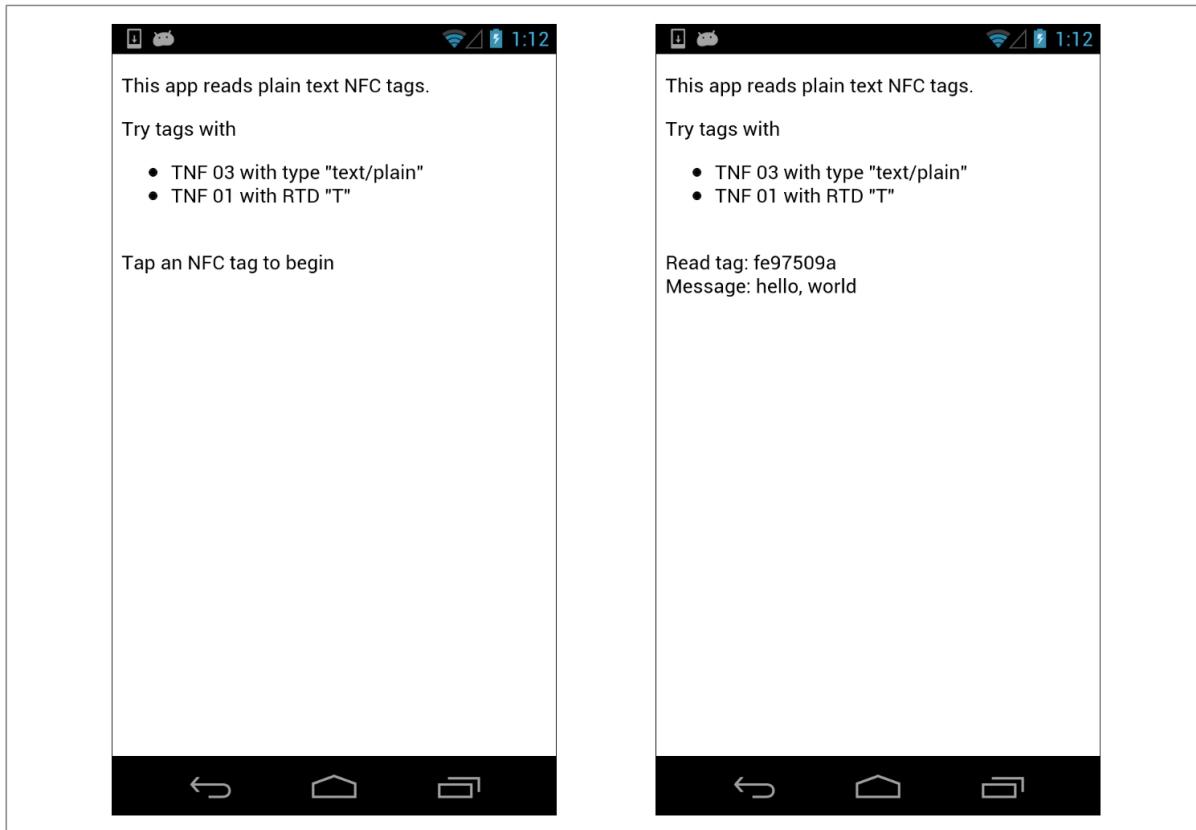


图 5-4：从 MimeReader 应用程序读取的结果：等待标签（左图）和读取带有 MIME 类型“text/plain”的文本标签（右图）

Android标签分发系统

到目前为止，你已经使用Android的前台调度系统捕捉到所有监听，PhoneGap-NFC插件在默认情况下就是如此使用的。换句话说，你的应用程序运行在前台，能接收到所有的NFC事件，然后决定哪些处理、哪些忽略。但是如果你的应用程序没有运行，它就得不到信息，这就不是一个好的交互设计了。

让我们想一想用户使用标签与另一个设备交互的理想顺序：她看到了标签，并将设备放过去，然后设备开始处理，她没有打开应用程序或者胡乱设置，手机的操作系统就已经知道去监听标签，然后它读取标签类型就去调用相应程序的相应activity。为了达到这个目的，标签识别必须发生在应用程序打开之前。其实，Android的标签分发系统通常也是这么工作的。

标签分发系统能够读取扫描到的标签，并根据它们的TNF和记录类型打开相应的应用程序，这个可以用intent filter做到。当Android成功读取TNF和记录类型后，它会试着将其映射到一个已知的MIME类型或者URI模式，这样它就能带着这些数据启动一个intent，然后它会扫描自己的应用数据库，看看哪个应用程序可以处理这个intent。如果不止一个应用程序可以处理，Android就会提供给用户去选择。在理想情况下，应用程序的NDEF信息应该是独特的，这样其他的应用程序就无法处理它们。intent filter都声明在程序的AndroidManifest.xml文件中。

要配置应用程序去监听某个特定的intent filter，则需要修改应用程序的platforms/android/AndroidManifest.xml文件，对于一个intent类型，同样需要添加一个文件到platforms/android/res/xml目录。你可以使用NDEF读取器项目试试多种组合看看这个应用程序会发生什么。开始时应用程序没有运行，当你将相关的标签轻触设备时，应用程序应该会打开。在此之前注意你的标签类型，这样你就知道哪些标签能触发应用程序，而哪些不能。

intent类型

有三种intent能让Android对标签做出回应。

TAG_DISCOVERED：这种类型的intent能被NFC读取器已知的任何类型执行，并且它也不是RFID或NFC标签类型之一。如果你使用的是任何兼容的RFID或NFC标签类型，那么一般不会看到这种intent。

下面是定义在XML中的代码：

```
<activity>
    ... other stuff in the activity element goes here

    <intent-filter>
        <action android:name="android.nfc.action.TAG_DISCOVERED"/>
    </intent-filter>
</activity>
```

TECH_DISCOVERED：当Android扫描到一个兼容的RFID或NFC标签时产生这种intent。为了使用这种intent，你必须在platforms/android/res/xml目录下的一个单独文件中声明你打算监听的标签类型的技术清单。例如，如果你打算监听NDEF可格式化的Mifare Ultralight标签，则需要配置intent filter来查找那些标签类型。首先，需要在AndroidManifest.xml中声明你正在使用action.TECH_DISCOVERED intent fileter：

```
<activity>
    ... other stuff in the activity element goes here

    <intent-filter>
        <action android:name="android.nfc.action.TECH_DISCOVERED"/>
    </intent-filter>

    <meta-data android:name="android.nfc.action.TECH_DISCOVERED"
```

```
    android:resource="@xml/nfc_tech_filter" />
</activity>
```

然后，需要在platforms/android/res/xml目录下创建一个名为nfc_tech_filter.xml的文件。具体代码如下：

```
<resources xmlns:xliff="urn:oasis:names:tc:xliff:document:1.2">
    <tech-list>
        <tech>android.nfc.tech.NdefFormattable</tech>
        <tech>android.nfc.tech.MifareUltralight</tech>
    </tech-list>
</resources>
```

这个技术清单（tech list）应该只指定你想过滤的类型。通过技术清单过滤后的标签都是可以匹配的标签。在这个例子中，只有Mifare Ultralight和NDEF可格式化的标签能触发这种intent。

可以指定多个技术清单，这样就能进行多组合的扫描。例如，你想要的标签可能是Mifare Ulralight和NDEF可格式化、 NdefA和NDEF可格式化，或者NdefB和NDEF可格式化。如此，技术清单文件看起来就像这样：

```
<resources xmlns:xliff="urn:oasis:names:tc:xliff:document:1.2">
    <tech-list>
        <tech>android.nfc.tech.NdefFormattable</tech>
        <tech>android.nfc.tech.MifareUltralight</tech>
    </tech-list>
    <tech-list>
        <tech>android.nfc.tech.NdefFormattable</tech>
        <tech>android.nfc.tech.NdefA</tech>
    </tech-list>
    <tech-list>
        <tech>android.nfc.tech.NdefFormattable</tech>
        <tech>android.nfc.tech.NdefB</tech>
    </tech-list>
</resources>
```

作为参考，下面给出包含所有可能的NFC相关技术类型的一个技术清单。这来自Android开发者网站的NFC基础页面（<http://bit.ly/nfc-basics>），你可以选择任何你需要的。

```
<resources xmlns:xliff="urn:oasis:names:tc:xliff:document:1.2">
    <tech-list>
        <tech>android.nfc.tech.IsoDep</tech>
        <tech>android.nfc.tech.NfcA</tech>
        <tech>android.nfc.tech.NfcB</tech>
        <tech>android.nfc.tech.NfcF</tech>
        <tech>android.nfc.tech.NfcV</tech>
        <tech>android.nfc.tech.Ndef</tech>
        <tech>android.nfc.tech.NdefFormattable</tech>
        <tech>android.nfc.tech.MifareClassic</tech>
        <tech>android.nfc.tech.MifareUltralight</tech>
    </tech-list>
</resources>
```

NDEF_DISCOVERED是第三种类型的intent filter，并且一般最常用。这是三种类型中最明确的。它会寻找一个NDEF格式标签，并能过滤一种特定的MIME类型或URL。下面是一种NDEF_DISCOVERED intent过滤MIME类型的text/plain：

```
<intent-filter>
    <action android:name="android.nfc.action.NDEF_DISCOVERED"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <data android:mimeType="text/plain" />
</intent-filter>
```

在Android中，带有TNF 01（Well-Known类型）的标签通常不会在一个intent fileter中指定。然而，有一些内置到操作系统的异常。带有TNF 01和T（文本）类型的RTD记录能被NDEF_DISCOVERED intent filter过滤去查找mimeType="text/plain"。所以，如果你要查找plain-text记录，使用TNF01、RTD“T”，然后像之前那样设置intent fileter。有了这样的组合，不管在前台还是后台过滤都能触发事件。

NDEF_DISCOVERED intent同样能让你过滤MIME类型和URL，甚至是指定类型的URL。例如，你还记得在第4章中格式化的标签嵌入有m.foursquare.com/venue的URL吗？下面是如何过滤它们的代码。

```
<intent-filter>
    <action android:name="android.nfc.action.NDEF_DISCOVERED"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <data android:scheme="http"
        android:host="m.foursquare.com"
```

```
    android:pathPrefix="/venue" />  
</intent-filter>
```

将这个添加到NDEF读取器应用程序的AndroidManifest.xml文件中，然后使用cordova run命令生成并部署应用程序。关闭它，然后将Foursquare标签之一轻触设备，你应该会看到一个选择器，要求你在NDEF读取器和Foursquare应用程序之间进行选择。为什么呢？因为它们都过滤这种URI模式。

更具体地设置intent filter，应用程序将会有更好的机会去响应它。鉴于此，想一下在第4章中为什么Samsung TecTiles程序和NFC Task Launcher程序使用它们的方式格式化标签。回想一下它们是如何格式化调用FourSquare的。

NFC Task Launcher（两条NDEF记录）：

记录1

- Record type: x/nfctl
- Payload: enZ: Foursquare; c: 4a917563f964a520401a20e3

记录2

- TNF: External
- Record type: android.com: pkg
- Payload: com.jwsoft.nfcactionlauncher

Samsung TecTiles：

记录1

- TNF: Well-Known
- y Record Type: U
- Payload: tactile: //www/samsung.com/tectiles

记录2

- TNF: Well-Known

- Record type: T
- Payload: ·enTask…Foursquare·com.joelapenna…

两个应用程序都过滤了唯一的MIME类型或者URI， NFC Task Launcher定义了自己的MIME类型（x/nfctl）。第二条记录是Andorid应用程序记录，它将启动NFC Task Launcher应用，然后使用包含在第一条记录里的有效载荷信息调用FourSquare。Samsung TecTiles定义了一个唯一的URI（tectile: //www/samsung.com/tectiles），所以Android会打开TecTiles， TecTile启动FourSquare的任务信息嵌入在第二条记录中。

Android应用记录

在Android 4.0版本（API 14）中引入了Android应用程序记录（Android Application Record，AAR），以此给开发者提供一种正确的方式来读取给定的标签。在action中你已经看到了Android应用程序记录；NFC Task Launcher例子使用了AAR。Android会扫描整个NDEF消息来查找一条包含AAR的NDEF记录，如果找到了，则处理该intent对应的应用，覆盖所有其他的调度优先级。如果你想让你的应用程序是唯一得到给定intent的，那么就要确保标签有一条AAR能让intent调用你的应用程序。覆盖AAR调度的唯一方式就是使用前台分发系统。

在前面的例子中，NFC Task Launcher的第二条记录是在Android 4.0之后添加的。第一条记录让标签分发系统来过滤对应的程序其实已经足够了，但是添加AAR可以更加确保这一点。此外，如果没有安装AAR涉及的应用程序，那么Android将打开Play Store并询问是否下载该应用程序。

总结

通过本章，你应该已经对怎样监听NDEF信息有了一个很好的理解。当应用程序在运行时，你可以使用前台分发系统，然后使用PhoneGap的事件监听器过滤不同类型的事件：发现标签、NDEF格式标签、NDEF信息或者带有特定MIME类型的NDEF消息。你也能使用后台分发系统让Android过滤相关的标签到你的应用程序。你可以使用action.TAG_DISCOVERED intent识别任何标签，使用action.TECH_DISCOVERED intent过滤指定的标签类型，使用action.NDEF_DISCOVERED intent过滤指定的MIME类型、记录类型或者URI模式。你也可以混合和匹配所有这些来过滤各种标签类型和内容。

NFC的一些应用可能使用多种方式去达到最终目标。在下一章中，将构建一个完整的基于NFC的应用，使用的都是你已经学过的知识。

第6章 一个NFC应用实践

到目前为止，你已经知道了NFC和NDEF之间的不同，并了解了NDEF消息和记录之间的关系。你也看到了Android的标签分发系统是如何在前台或后台通过类型过滤标签并做出响应的。在本章中，你将使用之前所学到的知识做一个完整的应用程序，通过这个应用程序使用移动设备来控制家里的灯光和音响。你要构建的这个应用程序是一个经典的例子，它在很多场合和科技会议上被介绍过。我们把它叫作“Mood Setter”应用。

这个应用程序的需求很简单：在家做各种事情时你可能需要不同的环境氛围。当你打扫房间时，你可能想要鲜艳漂亮的灯光和一些欢快的音乐。当你在书房办公桌前集中精力工作时，你可能会喜欢安静的音乐，或许不想让任何打扰来分散你的注意力，不需要点亮每个房间的灯，只要书房亮着就行。晚餐时，你可能希望有一种漂亮温暖的光芒围绕着桌子，有一些舒缓的背景音乐。抑或，你只想要一个轻松的夜晚、昏暗的灯光和一点Barry White的音乐。谁能抵挡这种诱惑？

为了实现这些功能，你需要在房子周围散布一些NFC标签，每个标签设置特定的参数信息，然后使用设备读取标签并控制灯光和音乐。在此应用中，灯是Philips Hue系统 (<https://www.meethue.com>)。Hue是一个照明系统，它结合了带有嵌入式ZigBee的LED灯具和一个特殊的以太网到ZigBee的Hub。Hub上运行有HTTP服务，设置灯光的命令是通过RESTful协议发送的JSON对象。这意味着你可以使用任何能发起HTTP请求的设备来控制Hue的灯光。可以在设备上播放音乐，同时将音频流通过蓝牙发送到立体声扬声器或立体声耳机。图6-1给出了硬件设置。

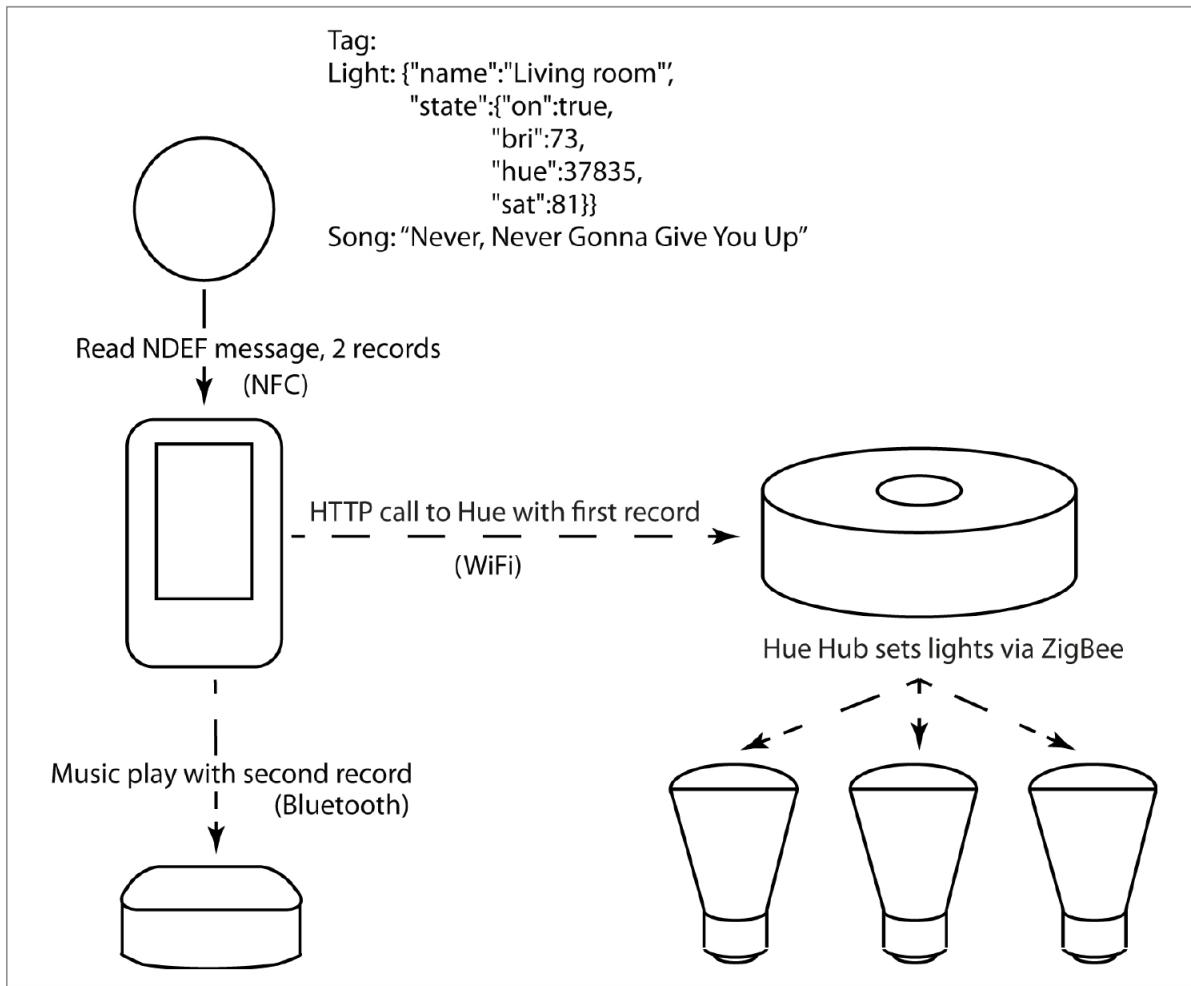


图6-1：Mood Setter应用程序系统图

蓝牙接收器可以是任何能接收蓝牙音频的音频设备。市场上有很多这类设备，其中有一些组合了放大器并内置蓝牙设备，而有一些则只是单纯地将蓝牙音频转换为模拟音频，使用它们时只要将其插到一个现有的普通音频接收器上即可。如果你愿意，甚至可以使用蓝牙耳机。我们选择Belkin的蓝牙音乐接收器 (<http://bit.ly/belkin-bmr>)，因为它便宜，而且可以插到我们现有的音响上。你也可以看看HomeSpot的具有NFC功能的蓝牙音频接收器 (<http://bit.ly/homespot-receiver>)，它可以通过NFC将设备与接收者配对。我们发现在这个应用程序里Belkin和HomeSpot都工作优异。一旦你的设备配上音频接收器，进入Android系统设置，选择“蓝牙”，连接到接收器，你就准备好了。



我们可以肯定，本书的读者不大可能会去买一个Philips Hue系统，但我们喜欢Philips使用Web标准（HTTP、REST和JSON）和开放无线协议（ZigBee的LightLink灯光控制协议）来实现灯光控制。Hue系统是一个很好的关于应用程序之间如何使用HTTP和JSON传递消息的例子。因此，此处所示的概念可以推广到其他基于Web的应用程序中。

用户交互

在使用这个应用程序前需要用户设置一些场景数据：Hue灯光的亮度、色调、饱和度、开或关；通过蓝牙选择哪首歌曲，播放、停止或暂停；读取或写入包含有音乐和灯光数据的NFC标签。

这个应用程序有两种主要的操作模式：读模式（通过读取标签里的设置数据自动改变灯光和音乐）和写模式（把当前设置保存到标签）。不管在哪种模式下，你都能控制歌曲和灯光。

在家当你轻触设备和标签时，如果应用程序在读模式下，它会自动改变灯光，并播放相应的音乐，具体的设定情况依赖标签里嵌入的设置信息。如果在写模式下，它会自动把当前设置写入标签。

该应用程序界面中有一个模式按钮用于切换读、写模式，有一个下拉菜单用来控制灯光，设置灯光的亮度、饱和度、色调、打开或关闭；另一个下拉菜单用来选择想要播放的歌曲，播放/暂停和停止歌曲。

该应用程序的用户界面如图6-2所示。它分为三个部分：顶部的标签模式部分、中间的灯光控制部分，以及底部的音乐控制部分。在HTML里，每个部分都有自己的div。在其他应用程序里用到的消息div，在这里也会用到，只是在你完成编码和测试后就会去掉。

当改变Hue灯光设置时，应用程序将在本地保存一份设置参数，以便靠近相应标签时将这些设置写入该标签。

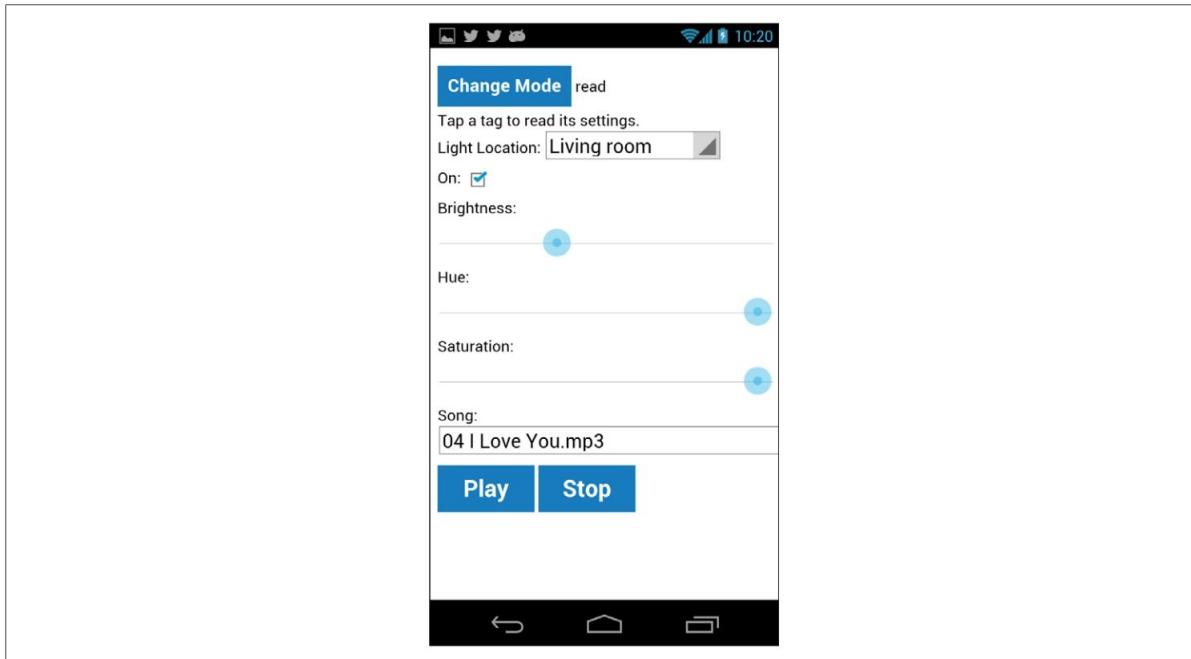


图6-2：Mood Setter应用程序用户界面

每个触发灯光和音乐的标签内都有一个含有两条记录的NDEF消息：灯光数据记录和音乐数据记录。灯光记录是一条自定义的MIME类型记录“text/hue”。因为它是一种自定义的MIME类型，所以很容易筛选。音乐记录是一种Well-Known的URI记录类型。当设备读取标签时，它会检查是否有text/hue记录，如果标签内有一条灯光记录，它将调用一个HTTP请求到Hue Hub用以控制灯光属性。如果发现标签上的URI记录，它会尝试播放记录里的歌曲文件。

为了写这个应用程序，你需要了解一些关于Hue平台的知识。

了解Hue

Philips Hue灯光系统是一个实施得当的网络家电例子。它使用Web标准实现端到端通信，使得应用程序开发人员可以扩展它的功能，而不必了解硬件。即使你对Hue系统本身不感兴趣，但本节将帮助你了解网络电器是如何通过互联网和其他设备互操作的。我们将指出Hue的哪些设计是开发应用程序时值得注意的地方。

Hue系统配有三个灯泡和一个Ethernet-to-ZigBee Hub，称为桥。你的设备使用基RESTful格式的HTTP请求与桥进行沟通。你可以从Meet Hue (<http://www.meethue.com/>) 下载符合Hue标准的默认应用程序，在那里你也会找到配置系统的指令（进一步说明，请参阅Hue开发者部分 (<http://developers.meethue.com/>)）。

对于任何无屏幕的互联网设备来说，首要问题就是如何让用户知道它已成功连接到网络，并开始控制它的客户端设备。Hue桥使用有线以太网连接到家庭路由器，使用DHCP方式获取IP地址，因此无须对WiFi网络做任何设置改动。通过两个交互步骤实现设备发现：第一步，你的设备和桥在同一网络时打开应用程序；第二步，按下桥上的“discovery”按钮。按下按钮几秒后，桥将收到应用程序发送的“discovery”响应。假设这两步在几秒钟内完成了，那么应用程序发现了桥并标识了自己，然后桥把该应用程序注册为一个有权控制灯光的应用程序。

具有NFC功能的HomeSpot蓝牙接收器的配对过程是值得体验一下的。HomeSpot接收器内有一个NFC标签，这个标签里包含了蓝牙配对的细节。通过轻触接收器，用户的具有NFC功能的手机就和HomeSpot配对上了。在通过无线方式进行蓝牙配对和Hue的发现过程前，我们必须先对这些设备硬件进行一些手工设置。不过，通常的网络设备不提供屏幕来进行设置操作。可以通过开发者代理页面 (<http://www.meethue.com/api/nupnp>) 找到你的Hue Hub的IP地址。这个页面将寻找局域网里的所有Hue Hub，然后给出IP地址。请注意，浏

览该页面的设备必须和Hub在同一局域网内。一旦获得了Hub的IP地址，你就可以通过浏览`http://your.hub.ip.address/debug/clip.html`直接连接到Hub上。你会看到一个页面，类似于图6-3。

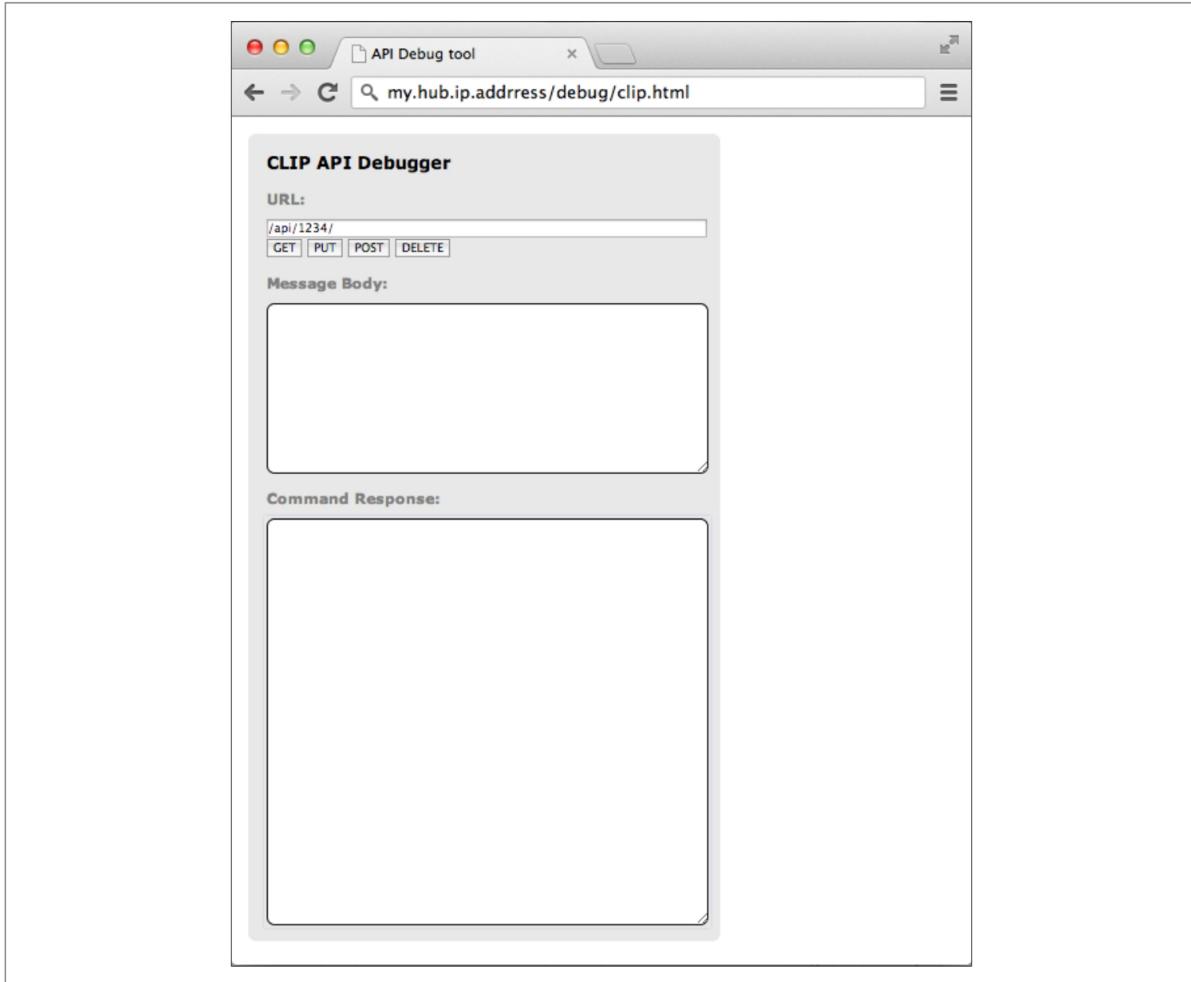


图6-3：Hue调试工具（这个页面适用于每一个Hue Hub，只要你知道Hub的IP地址）

一旦连接到Hue，你就需要为应用程序建立证书。证书通常是Hue应用程序建立的，但既然你自己写应用程序，那么就需要你自己来做这件事。这个过程有几步。首先，在调试工具的URL域输入以下内容：

/api/

然后将下面内容输入到body域，用你的实际应用程序的名称替换you rAppName，按下“link”按钮，并点击“POST”：

```
{"devicetype": "device type", "username": "yourAppName"}
```

你的设备类型可以是任何你想要的，如“Nexus 4”、“Galaxy SIII”或“My Magical Portal Window”。username至少10个字符，否则会报错。在这个应用程序中，将使用用户名“MoodSetterApp”。

你应该得到如下这样的响应：

```
[  
  {  
    "success": {  
      "username": "MoodSetterApp"  
    }  
  }  
]
```

现在你已经在Hue Hub上注册了一个用户名，可以控制它了。要看一下响应，在调试工具中输入以下内容：

```
/api/MoodSetterApp/
```

你会得到一个JSON格式的Hub上所有灯的描述信息。

Hue数据格式

为了知道怎样把数据写入标签，你需要了解Hue的数据格式。Hue数据采用JSON格式，这种格式易于读写。如果你正在开发自己的互联网设备应用程序，这种格式值得你去研究一下。在第7章中，你将开发一个具有NFC功能的门锁，它使用的也是JSON数据格式。

Philips有很好的开发者参考文档 (<http://developers.meethue.com/>)，清楚地解释了所有API。对于这个应用程序，你需要知道的是，每个灯在JSON对象里的描述如下：

```
"1":  
{"state":  
 {"on":true,"bri":65,"hue":44591,"sat":254,"xy":[0.1953,0.0959],  
 "ct":500,"alert":"none","effect":"none","colormode":"hs","reachable":true},  
 "type":"Extended color light",  
 "name":"Living room",  
 "modelid":"LCT001",  
 "swversion":"65003148",  
 "pointsymbol":  
 {"1":"none","2":"none","3":"none","4":"none",  
 "5":"none","6":"none","7":"none","8":"none"}  
}
```

有很多属性，但是你只需修改其中的一些属性。这是Hue数据协议值得学习的另一个方面：接收数据的设备（桥）能接受整个消息的任何有效的子集，如果有效，就起作用。

对于每个灯，你需要改动4个属性：开关、亮度、色调和饱和度。下面是改变灯的JSON格式例子：

```
"1":          // The light number  
{"state":      // the light's state, including:  
 {"on":true,    // whether it's on or off  
 "bri":65,    // the brightness (0 - 254)  
 "hue":44591, // the hue (0 - 65280 in the CIE colorspace)  
 "sat":254},  // the saturation of the color (0 - 254)
```

```
    }  
}
```

你可能想修改灯的名称，以反映它们具体位于房子的那个地方。这时你需要在调试工具中输入以下命令：

URL:

```
/api/MoodSetterApp/lights/1/
```

Body:

```
{"name": "First Light"}
```

得到的响应应该如下：

```
[  
  {  
    "success": {  
      "/lights/1/name": "First Light"  
    }  
  }  
]
```

改变灯的状态，可以这样做：

URL:

```
/api/MoodSetterApp/lights/1/state
```

Body:

```
{"on": true, "bri": 65, "hue": 44591, "sat": 254}
```

每个属性的改变你都应该得到一个成功与否的反馈（另一个很好的通用原则），现在灯的颜色应该是愉快的蓝紫色。

实际上，使用REST API不可能一次发送整个JSON对象。你必须逐个灯地设置属性。

假设你不熟悉灯光色彩理论，也不懂色调和饱和度属性。Philips采用的是被称为CIE的色彩体系，该体系设计于1931年，由国际照明委员会（CIE）制定用来统一描述光颜色的映射图。简单地说，CIE标准把颜色映射为两部分：亮度和色度。亮度指的是颜色在黑色到白色之间的相对值，而色度是指颜色的相对值。你可以这么想：黑、白、灰都具有相同的色度，但各有不同的亮度。图6-4显示了CIE颜色空间的色度图。

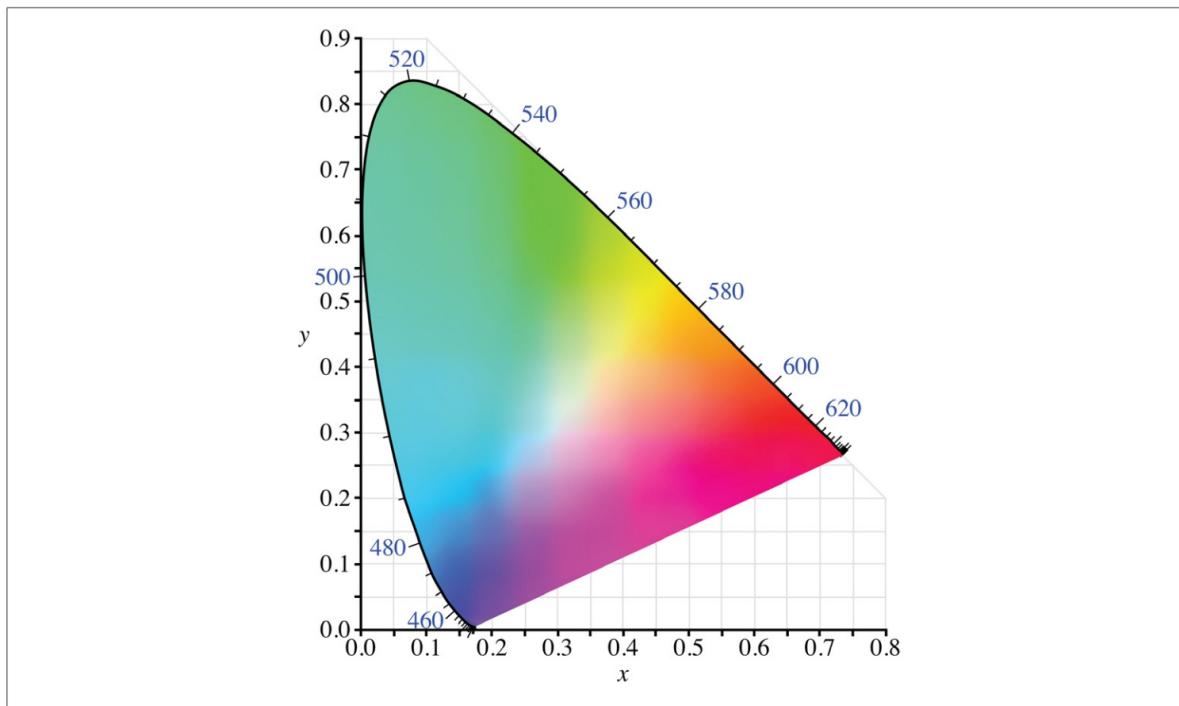


图6-4：CIE颜色空间的色度图（图片源自维基百科）

当你改变一个灯的色调时，将会看到它在CIE颜色空间上的变化，从低端的红色移动到中间的其他颜色，再回到红色。当你改变饱和度时，将会看到颜色从一个充满活力的颜色褪色到苍白色。

Hue Hub控制3个灯，你可以发送一个JSON对象，像这样：

```
{"1":  
  {"state":
```

```
        {"on":true,"bri":65,"hue":44591,"sat":254}  
    },  
    "2":  
        {"state":  
            {"on":true,"bri":254,"hue":13122,"sat":211}  
        },  
    "3":  
        {"state":  
            {"on":true,"bri":255,"hue":14922,"sat":144}  
        }  
    }  
}
```

此JSON对象将是本应用程序写入NFC标签的两条NDEF记录中的第一条记录的载荷。正如前面提到的，这条记录类型为MIME类型text/hue。

Hue的REST API

从调试工具你可能已经注意到了，Hue的请求API是RESTful。为了获取数据，你可以使用格式：/api/MoodSetterApp/，然后添加你想要的细节，并发出HTTP GET请求。例如，为了得到灯1的状态，就用/api/MoodSetterApp/lights/1/state。要更改数据，则使用相同格式的一个PUT请求。在这个应用程序中，你要构建一个HTTP请求到Hub获取数据，一个请求用来将新数据PUT到Hub，像这样：

```
url: 'http://' + hub.ipaddress + '/api/' + hub.username  
    + '/lights/' + lightId + '/state',
```

下面这个请求从Hub获取数据：

```
url: 'http://' + hub.ipaddress + '/api/' + hub.username,
```

因为API就是REST URL和JSON，所以你可以通过使用zepto.js——一个流行的jQuery库精简版来简化应用程序。在“用户接口（UI）”一节你会看到HTML头。

Android shell

另一条NDEF记录是一条URI记录，它内含播放歌曲的完整文件路径。该歌曲将用PhoneGap媒体API播放。为了简单化，你把所有想要听的歌曲都放在一个目录中，或者使用默认目录。

你正在使用的Android Debug Bridge（ADB），不仅可以告诉你调试时的日志信息，还可以用来从计算机命令行接入Android设备。为了获得shell，在计算机上打开一个终端窗口，将Android设备通过USB连接到计算机，并输入以下命令：

```
$ adb shell
```

你现在可以直接访问该设备了，而且可以让shell做很多期待的事情：用ls命令列出文件，用cd命令切换目录，用mkdir命令新建目录，等等。如果列出设备根目录，你将会看到一个叫作sdcard的目录。这是Android设备上用户文件的默认存储空间，它是一个存放歌曲文件的好地方。

这个应用程序需要一些歌曲，所以在设备上新建一个存放歌曲的目录。在Terminal输入：

```
$ adb shell  
shell@android:/ $ mkdir sdcard/myMusic  
shell@android:/ $ exit
```

现在退出adb shell，回到计算机的shell界面，在计算机上找一些歌曲，传到设备上，转到歌曲所在的目录，在Terminal键入：

```
$ adb push songName.mp3 /sdcard/myMusic/.
```

这将把文件从计算机传到设备上。你可以重新登录到设备，用adb shell检查文件是否已经传入。

PhoneGap媒体API

要实现在这个应用程序里播放音乐，使用PhoneGap媒体API是比较好的选择，它允许你录音和播放音频。在这个应用程序里，只用它来播放歌曲，你可以在PhoneGap Guides网站

(<http://docs.phonegap.com>) 的“Media”栏目下找到全部API的说明文档。媒体API通过创建一个新的媒体播放对象来初始化，像如下这样：

```
var playBack = new Media(source, success, error, status);
```

媒体对象的参数可以让你控制它的行为。源URL可以是本地文件，使用格式为file: //path/to/filename，也可以使用远程URL，格式为http://example.com/path/to/filename。success、error和status处理程序都是可选的。媒体对象已经完成了当前播放、录音或者停止后，success处理程序被调用。当有一个播放错误时error处理程序被调用，而当改变播放状态时status处理程序被调用。status处理程序是很有用的，它会判断现在的状态是暂停还是播放中，来决定按钮控件的操作行为是继续播放还是停止。

媒体对象有一些播放控制的通用功能：启动、停止或暂停播放；获取当前播放位置或歌曲的播放时长；跳到一个特定的位置；释放操作系统的音频资源；开始和停止录音。其中的一些功能在下面的代码中将被使用。

为了使用媒体API，需要安装两个插件：Media插件和Filesystem插件。用如下命令安装这两个插件：

```
$ cordova plugin add \
  https://git-wip-us.apache.org/repos/asf/cordova-plugin-media.git
$ cordova plugin add \
  https://git-wip-us.apache.org/repos/asf/cordova-plugin-file.git
```

你还需要修改应用程序的AndroidManifest.xml文件，添加下列权限：

```
<uses-permission android:name="android.permission.RECORD_AUDIO" />
<uses-permission android:name="android.permission.MODIFY_AUDIO_SETTINGS"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

此外，应用程序的config.xml文件
(platforms/android/res/xml/config.xml) 必须包含以下插件：

```
<plugin name="Media" value="org.apache.cordova.AudioHandler" />
```

在默认情况下，所有的这些XML设置在你使用cordova create命令时会自动添加。

用户接口 (UI)

现在你已经知道了这个项目中所使用的新工具，是时候来创建项目，开始写代码了。

为此，你需要前面章节中的工具，以及一些新的硬件：

- 具有NFC功能的Android设备
- Android软件开发包（SDK）
- 一个文本编辑器
- 计算机上安装有Cordova CLI
- 安装Node.js和NPM
- Zepto jQuery库精简版，Zepto JS库 (<http://zeptojs.com/>) 精简版也行
- 各类NFC标签（为达到跨设备使用的效果，最好使用NFC论坛标签类型，可参见“设备与标签类型匹配”一节来了解标签和设备的适配情况）
- Philips Hue照明系统
- 蓝牙音频接收器，如Belkin蓝牙音乐接收器或HomeSpot具有NFC功能的蓝牙音频接收器

开始创建一个新项目：

```
$ cordova create ~/MoodSetter com.example.moodsetter MoodSetter ❶
$ cd ~/MoodSetter❷
$ cordova platform add android
$ cordova plugin add https://github.com/don/phonegap-nfc
$ cordova plugin add \
  https://git-wip-us.apache.org/repos/asf/cordova-plugin-media.git
$ cordova plugin add \
  https://git-wip-us.apache.org/repos/asf/cordova-plugin-file.git
```

❶Windows用户用%userprofile%\MoodSetter替换~/MoodSetter。

②Windows用户用/d%UserProfile%\MoodSetter替换~/MoodSetter。

把下载的Zepto JS库精简版和zepto.min.js文件复制到www/js目录下。

当项目准备好后，打开www目录中的index.html和index.js文件。

下面是用户界面的HTML文件（用户界面见图6-2）。在顶部你要添加一些CSS样式，使按钮和滑动条更容易点击：

```
<!DOCTYPE html>
<html>
  <head>
    <title>Mood Setter</title>
    <style>
      input {
        margin-top: 10px;
        margin-bottom: 10px;
      }
      input[type='range'] { /* finger sized sliders */
        display: block;
        width: 100%;
      }
      .button {
        display: inline-block;
        padding: 10px;
        min-width: 80px;
        margin-top: 10px;
        margin-bottom: 5px;
        text-align: center;
        background-color: #177bbd;
        color: #ffffff;
      }
      select {
        font-size: 1.25em;
      }
    </style>
  </head>
  <body>
    <div class="app">
      <div id="tagMode">
        <h3 id="modeButton" class="button">Change Mode</h3>
        <span id="modeValue"></span><br />
        <span id="tagModeMessage"></span>
      </div>
    </div>
```

```

<div id="lightControl">
    Light Location:
    <select id="lightNumber">
        <option value="1" >No Lights Found</option>
        <option value="2" >No Lights Found</option>
        <option value="3" >No Lights Found</option>
    </select><br />
    <label for="lightOn">On:</label>
    <input type="checkbox" id="lightOn"><br />
    Brightness: <input type="range" id="bri" min="0" max="254">
    Hue: <input type="range" id="hue" min="0" max="65535">
    Saturation: <input type="range" id="sat" min="0" max="254">
</div>
<div id="songPicker">
    Song:
    <div>
        <select id="songs">
            <option>Loading Songs</option>
        </select>
    </div>
    <div>
        <h2 id="playButton" class="button">Play</h2>
        <h2 id="stopButton" class="button">Stop</h2>
    </div>
</div>
<div id="messageDiv">No message</div>
</div>
<script type="text/JavaScript" src="js/zepto.min.js"></script>
<script type="text/JavaScript" src="cordova.js"></script>
<script type="text/JavaScript" src="js/index.js"></script>
<script>
    app.initialize();
</script>
</body>
</html>

```

这个应用程序（div）有4个主要部分：

- mode部分，其中包含模式按钮，并显示当前模式
- lightControl部分，其中包含所有对灯的控制
- songPicker部分，可以让你选择和控制歌曲
- message部分，是通用信息部分

在index.js文件中你会看到，所有的用户控件都有监听器。

在HTML文件中灯光控件的名称，和Hue的JSON格式里的属性名称是一致的。每个对象的取值范围就是Philips Hue规范中规定的范围。初始值由Hub决定。在lightNumber下拉菜单里灯光的名称是动态变化的。

应用程序代码

应用程序的主要代码通常在index.js文件中。完整的应用程序源代码可以在GitHub (<http://bit.ly/moodsetter>) 上找到。然而，这个JavaScript程序的开始点与通常的程序有所不同。通常程序开始时会定义一些应用程序变量，而这个程序首先定义了一个变量来保存Hub的所有属性，如下：

```
var hub = {           // a copy of the hue settings
  lights: {},        // states & names for the individual lights
  ipAddress: null,   // IP address of the hue
  appTitle: "NFC Mood Setter", // The App name
  username: "MoodSetterApp", // the App's username
  currentLight: 1     // the light you're currently setting
};
```

你使用这个对象来存储从Hub获取到的设置信息，以及应用程序的标题、你的用户名和目前你正在用户界面上操纵的灯。实际上，当你查询Hub时，Hue回复的JSON格式响应信息的内容非常丰富，所以你只要提取出感兴趣的就行了。

接下来是应用程序变量，一开始定义了几个全局参数：

```
var app = {
  mode: 'write',      // the tag read/write mode
  mimeType: 'text/hue', // the NFC record MIME Type
  musicPath: 'file:///sdcard/myMusic/', // path to your music
  songPlaying: null,    // media handle for the current song playing
  songTitle: null,      // title of the song
  musicState: 0,        // state of the song: playing stopped, etc.
  songUri: null,
```

通用功能

下面是initialize () 和bindEvents () 函数。这里的bindEvents () 函数初始化所有控件的监听器，以及监听暂停和恢复事件：

```
/*
  Application constructor
*/
initialize: function() {
  this.bindEvents();
  console.log("Starting Mood Setter app");
},
/*
  binds events that are required on startup to listeners.
*/
bindEvents: function() {
  document.addEventListener('deviceready', this.onDeviceReady, false);
  // hue faders from the UI: brightness, hue, saturation:
  bri.addEventListener('touchend', app.setBrightness, false);
  hue.addEventListener('touchend', app.setHue, false);
  sat.addEventListener('touchend', app.setSaturation, false);
  lightOn.addEventListener('change', app.setLightOn, false);
  lightNumber.addEventListener('change', app.getHueSettings, false);
  // buttons from the UI:
  modeButton.addEventListener('click', app.setMode, false);
  songs.addEventListener('change', app.onSongChange, false);
  playButton.addEventListener('touchstart', app.toggleAudio, false);
  stopButton.addEventListener('touchstart', app.stopAudio, false);
  // pause and resume functionality for the whole app:
  document.addEventListener('pause', this.onPause, false);
  document.addEventListener('resume', this.onResume, false);
},
```

在onDeviceReady () 处理程序中，你要使用findControllerAddress () 函数到Hue Hub进行初始查询。你要设置应用程序的读/写模式。你也设置了三个NFC监听器，前两个将监听任何NFC格式标签或NDEF标签，以便在写模式下可以写入；第三个将监听MIME类型，尤其是text/hue类型。任何包含MIME类型记录的标

签，在理想情况下，其第一条记录用于设置灯光，第二条记录用于播放音乐。在MIME媒体处理程序中，你需要区分读模式还是写模式。

最后以调用getSongs（）函数结尾。它查找你所指定的音乐目录，并把歌曲的名字和URI填充在HTML的selector里：

```
/*
  runs when the device is ready for user interaction.
*/
onDeviceReady: function() {
    app.clear();          // clear any messages onscreen
    app.findControllerAddress(); // find address and get settings
    app.setMode();         // set the read/write mode for tags
    // listen for NDEF Formattable tags (for write mode):
    nfc.addNdefFormattableListener(
        app.onWritableNfc,      // tag successfully scanned
        function (status) {     // listener successfully initialized
            console.log("Listening for NDEF-formattable tags.");
        },
        function (error) {       // listener fails to initialize
            app.display("NFC reader failed to initialize " +
                JSON.stringify(error));
        }
    );
    // listen for NDEF tags so you can overwrite MIME message onto them
    nfc.addNdefListener(
        app.onWritableNfc,      // NDEF type successfully found
        function() {             // listener successfully initialized
            console.log("listening for Ndef tags");
        },
        function(error) {         // listener fails to initialize
            console.log("ERROR: " + JSON.stringify(error));
        }
    );
    // listen for MIME media types of type 'text/hue' (for read or write)
    // Android calls the most specific listener, so text/hue tags end up here
    nfc.addMimeTypeListener(
        app.mimeType,           // what type you're listening for
        app.onMimeMediaNfc,     // MIME type successfully found
        function() {             // listener successfully initialized
            console.log("listening for mime media tags");
        },
        function(error) {         // listener fails to initialize
            console.log("ERROR: " + JSON.stringify(error));
        }
    );
    app.getSongs();          // load the drop-down menu with songs
},
```

在onDeviceReady () 后，添加display () 和clear () ，用于在 message div中显示消息：

```
/*
    appends @message to the message div:
*/
display: function(message) {
    var textNode = document.createTextNode(message),
        lineBreak = document.createElement("br"); // a line break

    messageDiv.appendChild(lineBreak);           // add a line break
    messageDiv.appendChild(textNode);            // add the message node
},

/*
    clears the message div:
*/
clear: function() {
    messageDiv.innerHTML = "";
},
```

全局事件处理程序

接下来是第一个事件处理程序，`onPause ()`、`onResume ()` 和 NFC事件处理程序。这些事件是仅有的需要监听的非用户控件事件。这些事件处理程序是很基本的处理程序。前两项检查媒体播放的状态，然后根据状态来决定切换成暂停还是继续播放。

```
/*
    This is called when the app is resumed
*/
onResume: function() {
    if (app.musicState === Media.MEDIA_PAUSED) {
        app.startAudio();
    }
},
/*
    runs when an NdefListener or NdefFormattableListener event occurs.
*/
onWritableNfc: function(nfcEvent) {
    if (app.mode === "write") {
        app.makeMessage(); // in write mode, write to the tag
    }
},
```

NFC事件处理程序只处理NFC事件。`onWritableNfc`处理程序是`NdefListener`和`NdefFormattableListener`的处理程序。当在写模式时，可格式化监听器让你写可以格式化的任何空白标签，更通用的`NdefListener`让你用MIME媒体类型覆盖现有的NFC标签。举例来说，你有一个之前应用程序使用的标签，并且那个应用程序也使用MIME类型，但不是`MimeTypeListener`监听对象之一，`NdefListener`就会捕获它。

只有当MIME类型为`text/hue`的标签出现时，`onMimeMediaNfc ()`处理程序才会被调用。如果在读模式下，它会读取标签，并尝试改变音乐和灯光。如果在写模式下，它会覆盖标签：

```
/*
  runs when an NdefListener or NdefFormattableListener event occurs.
*/
onWritableNfc: function(nfcEvent) {
  if (app.mode === "write") {
    app.makeMessage(); // in write mode, write to the tag
  }
},
/*
  runs when a MimeMedia event occurs.
*/
onMimeMediaNfc: function(nfcEvent) {
  var tag = nfcEvent.tag;
  if (app.mode === "read") { // in read mode, read the tag
    // when app is launched by scanning text/hue tag
    // you need to add a delay so the call to get the
    // hub address can finish before you call the api.
    // if you have the address, delay 0, otherwise, delay 50:
    var timeout = hub.ipaddress ? 0 : 500;
    setTimeout(function() {
      app.readTag(tag);
    }, timeout);
  } else { // if you're in write mode
    app.makeMessage(); // in write mode, write to the tag
  }
},
```

既然标签操作方式取决于所在的模式，那以你还需要一个模式处理程序。这是模式按钮的事件处理程序。你可以看到它在之前的bindEvents () 函数里初始化。它只是改变app.mode变量，并显示在标签模式的div里，以便用户知道它处于什么模式下：

```
/*
  Sets the tag read/write mode for the app.
*/
setMode: function() {
  if (app.mode === "write") { // change to read mode
    app.mode = "read";
    tagModeMessage.innerHTML = "Tap a tag to read its settings.";
  } else { // change to write mode
    app.mode = "write";
    tagModeMessage.innerHTML = "Tap a tag to write current settings.";
  }
  modeValue.innerHTML = app.mode; // set text in the UI
},
```

Hub通信功能

一旦基本的初始化工作完成后，你需要查询Hub，得到所需要的设置信息，并将它们复制到Hub对象中。查询是用Zepto JS库中的jQuery ajax函数。这里显示的findControllerAddress () 函数，创建了一个Ajax对象，该对象到http://www.meethue.com/api/nupnp去查询本地局域网上是否有任何Hub的IP地址。如果找到了，则设置hub.ipaddress。如果没有找到，则提示用户没有发现Hub：

```
/*
 * Find the Hue controller address and get its settings
 */
findControllerAddress: function() {
    $.ajax({
        url: 'http://www.meethue.com/api/nupnp',
        dataType: 'json',
        success: function(data) {
            // expecting a list with a property called internalipaddress:
            if (data[0]) {
                hub.ipaddress = data[0].internalipaddress;
                app.getHueSettings(); // copy the Hue settings locally
            } else {
                navigator.notification.alert(
                    "Couldn't find a Hue on your network");
            }
        },
        error: function(xhr, type){ // alert box with the error
            navigator.notification.alert(xhr.responseText +
                " (" + xhr.status + ")", null, "Error");
        }
    });
},
```



如果网络里没有DHCP服务，或者Hue和Android设备在不同的网络上运行，那么你必须修改这个程序。Hue需要一个公网IP地址，或者你需要在和Hue连接的路由器上打开端口映射功能。设置好后，在应用程序开始时，输入Hue的IP地址到hub.ipaddress，并注释掉findControllerAddress（里的下面这行）：

一旦你做了修改，应用程序将直接用你给的固定地址寻找Hub。

发现Hub后，需要在上面注册你的用户名和应用程序名称。下面是函数ensureAuthorized（）和authorize（）。ensureAuthorized（）只检查Hub，看看你的用户名和应用程序名称是否已在上面成功注册。如果失败，则调用authorize（），并尝试获取授权。失败时提示用户，这时你需要按Hub的“link”按钮，以授权新用户和应用程序：

```
/*
 * Checks that the username is authorized for this hub.
 */
ensureAuthorized: function() {
    var message; // response from the hub
    // query the hub:
    $.ajax({
        type: 'GET',
        url: 'http://' + hub.ipaddress + '/api/' + hub.username,
        success: function(data){ // successful reply from the hub
            if (data[0].error) {
                // if not authorized, users gets an alert box
                if (data[0].error.type === 1) {
                    message = "Press link button on the hub.";
                } else {
                    message = data[0].error.description;
                }
                navigator.notification.alert(message,
                    app.authorize, "Not Authorized");
            }
        },
        error: function(xhr, type){ // error message from the hub
            navigator.notification.alert(xhr.responseText +
                " (" + xhr.status + ")", null, "Error");
        }
    });
}, /*
```

```

    Authorizes the username for this hub.
*/
authorize: function() {
    var data = {                      // what you'll send to the hub:
        "devicetype": hub.appTitle,   // device type
        "username": hub.username     // username
    },
    message;                      // reply from the hub
    $ajax({
        type: 'POST',
        url: 'http://' + hub.ipaddress + '/api',
        data: JSON.stringify(data),
        success: function(data){    // successful reply from the hub
            if (data[0].error) {
                // if not authorized, users gets an alert box
                if (data[0].error.type === 101) {
                    message = "Press link button on the hub, then tap OK.";
                } else {
                    message = data[0].error.description;
                }
                navigator.notification.alert(message,
                    app.authorize, "Not Authorized");
            } else {                  // if authorized, give an alert box
                navigator.notification.alert("Authorized user " +
                    hub.username);
                app.getHueSettings(); // if authorized, get the settings
            }
        },
        error: function(xhr, type){ // error reply from the hub
            navigator.notification.alert(xhr.responseText +
                " (" + xhr.status + ")", null, "Error");
        }
    });
},

```

一旦注册到Hub上，你需要从它那里得到设置信息，并将它们复制到本地的Hub对象中。findHueSettings（和authorize（）会调用下面的函数。它不会复制所有的Hub设置信息，只需要开/关、亮度、色调和饱和度的设置信息：

```

/*
Get the settings from the Hue and store a subset of them locally
in hub.lights. This is for both setting the controls, and for
writing to tags:
*/

```

```

getHueSettings: function() {
    // query the hub and get its current settings:

    $ajax({
        type: 'GET',
        url: 'http://' + hub.ipaddress + '/api/' + hub.username,
        success: function(data) {
            if (!data.lights) {
                // assume they need to authorize
                app.ensureAuthorized();
            } else {
                // the full settings take more than you want to
                // fit on a tag, so just get the settings you want:
                for (var thisLight in data.lights) {
                    hub.lights[thisLight] = {};
                    hub.lights[thisLight].name = data.lights[thisLight].name;
                    hub.lights[thisLight].state = {};
                    hub.lights[thisLight].state.on =
                        data.lights[thisLight].state.on;
                    hub.lights[thisLight].state.bri =
                        data.lights[thisLight].state.bri;
                    hub.lights[thisLight].state.hue =
                        data.lights[thisLight].state.hue;
                    hub.lights[thisLight].state.sat =
                        data.lights[thisLight].state.sat;
                }
                app.setControls();
            }
        }
    });
},

```

你刚刚编写的函数，只是获取设置信息，那么如何修改Hub上的设置信息呢？你需要一个函数，让用户能随时控制灯光。下面是putHueSettings ()，它接受一个JSON对象，并将其转换为字符串，发送到Hub：

```

/*
    sends settings to the Hue hub.
*/
putHueSettings: function(settings, lightId) {
    // if no lightId is sent, assume the current light:
    if (!lightId) {
        lightId = hub.currentLight;
    }
}

```

```
// if the light's not on, you can't set the other properties.  
// so delete the other properties before sending them.  
// if "on" is a property and it's false:  
if (settings.hasOwnProperty("on") && !settings.on) {  
    // go through all the other properties:  
    for (var prop in settings) {  
        // if this property is not inherited:  
        if (settings.hasOwnProperty(prop)  
            && prop != "on") {    // and it's not the "on" property  
            delete(settings[prop]); // delete it  
        }  
    }  
}  
// set the property for the light:  
$.ajax({  
    type: 'PUT',  
    url: 'http://' + hub.ipaddress + '/api/' + hub.username +  
        '/lights/' + lightId + '/state',  
    data: JSON.stringify(settings),  
    success: function(data){  
        if (data[0].error) {  
            navigator.notification.alert(JSON.stringify(data),  
                null, "API Error");  
        }  
    },  
    error: function(xhr, type){  
        navigator.notification.alert(xhr.responseText + " (" +  
            xhr.status + ")", null, "Error");  
    }  
});  
},
```

该函数不关心设置了什么，也不更新本地的Hub对象。这个函数把所有的信息都丢给调用它的函数来处理。但它处理了一个重要的错误：如果灯没打开，Hue不让你设定该灯的色调、亮度和饱和度。所以这个函数会检查正在发送的设置是否是开灯属性。如果是开灯属性，那么将删除其他属性的设置，只发送开灯设置。

用户界面控件事件处理程序

下一组功能是用户界面的控件事件处理程序。

通过getHueSettings () 得到Hub设置后，可以相应地设置用户界面控件来匹配它们当前的状态。在getHueSettings () 末尾调用下面的函数setControls () ，来更新UI控件的值：

```
/*
 Set the value of the UI controls using the values from the Hue:
 */
setControls: function() {
    hub.currentLight = lightNumber.value;

    // set the names of the lights in the dropdown menu:
    // (in a more fully developed app, you might generalize this)
    lightNumber.options[0].innerHTML = hub.lights["1"].name;
    lightNumber.options[1].innerHTML = hub.lights["2"].name;
    lightNumber.options[2].innerHTML = hub.lights["3"].name;

    // set the state of the controls with the current choice:
    var thisLight = hub.lights[hub.currentLight];
    hue.value = thisLight.state.hue;
    bri.value = thisLight.state.bri;
    sat.value = thisLight.state.sat;
    lightOn.checked = thisLight.state.on;
},
```

用户界面的灯光控件是由一系列事件处理程序控制的，每个灯光控件用来设定灯的特定属性。图6-5显示了灯光控件的细节。下拉菜单的事件处理程序是getHueSettings () ，它获取当前设置，并更新其他控件。

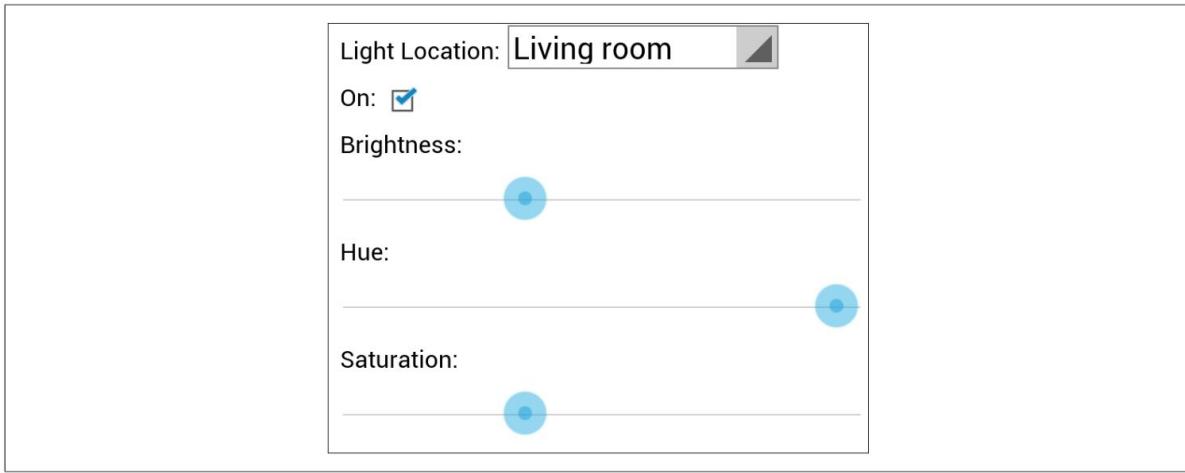


图6-5：应用程序的灯光控件界面

On、Brightness、Hue和Saturation控件适用于任何在Light Location下拉菜单里选择的灯。每一个事件处理程序会从控件获取最新值，然后设置本地的Hue对象。最后，每个处理程序都会使用新的设置来更新Hue Hub：

```
/*
These functions set the properties for a Hue light:
Brightness, Hue, Saturation, and On State
*/
setBrightness: function() {
    // get the value from the UI control:
    var thisBrightness = parseInt(bri.value, 10);
    // get the property from hub object:
    var thisLight = hub.lights[hub.currentLight];
    // change the property in hub object:
    thisLight.state.bri = thisBrightness;
    // update Hue hub with the new value:
    app.putHueSettings({ "bri": thisBrightness });
},
setHue: function() {
    // get the value from the UI control:
    var thisHue = parseInt(hue.value, 10);
    // get the property from hub object:
    var thisLight = hub.lights[hub.currentLight];
    // change the property in hub object:
    thisLight.state.hue = thisHue;
    // update Hue hub with the new value:
    app.putHueSettings( { "hue": thisHue } );
},
```

```
setSaturation: function() {
  // get the value from the UI control:
  var thisSaturation = parseInt(bri.value, 10);
  // get the property from hub object:
  var thisLight = hub.lights[hub.currentLight];
  // change the property in hub object:
  thisLight.state.sat = thisSaturation;
  // update Hue hub with the new value:
  app.putHueSettings({ "sat": thisSaturation });
},
setLightOn: function() {
  // get the value from the UI control:
  var thisOn = lightOn.checked;
  // get the property from hub object:
  var thisLight = hub.lights[hub.currentLight];
  // change the property in hub object:
  thisLight.state.on = thisOn;
  // update Hue hub with the new value:
  app.putHueSettings( { "on": thisOn } );
},
```

这些事件处理程序确保UI控件、本地Hue对象和远程Hue Hub的设置信息同步。

当设计标签读写功能时，你需要一个函数，用它来读取标签里的所有灯光的属性，并将其写到Hue Hub。你可以把所有灯的状态当作一个参数传给这个函数。把这个函数和之前的函数组合起来：

```
/*
 Sets the state for all the lights. Assumes it's getting
 a JSON object like this:
 {
   "1": {"state": {"on":true,"bri":65,"hue":44591,"sat":254}},
   "2": {"state": {"on":true,"bri":254,"hue":13122,"sat":211}},
   "3": {"state": {"on":true,"bri":255,"hue":14922,"sat":144}}
 }
*/
setAllLights: function(settings) {
  for (var thisLight in settings) {
    app.putHueSettings(settings[thisLight].state, thisLight);
  }
},
```

音乐界面事件处理程序

用户界面灯光控件介绍完了，是时候来编写音乐播放事件处理程序了。图6-6显示了这些控件的细节。



图6-6：应用程序的音乐控件界面

第一个音乐事件处理程序是getSongs（）。这个处理程序被onDeviceReady（）调用。它使用PhoneGap的文件API来读取音乐目录下的目录清单，并用得到的所有歌曲名称和URI来填充UI里的音乐selector。

这个功能在很大程度上依赖于回调，所以阅读代码时从底部开始读比较好。它首先用resolveLocalFileSystemURI（）查找目录，如果成功，则调用foundDirectory（）。然后查找目录条目，如果成功，则调用foundFiles（）填写到HTML中的selector。如果失败，则有一个常见故障处理程序给出提示：弹出一个警告框，提示失败的原因。

```
/*
  gets a list of the songs in your music directory and
  populates an options list in the UI with them
*/
getSongs: function() {
  // failure handler for directoryReader.readEntries():
  var failure = function(error) {
    alert("Error: " + JSON.stringify(error));
  };
  // success handler for directoryReader.readEntries():
  var foundFiles = function(files) {
```

```

if (files.length > 0) {
    // clear existing songs
    songs.innerHTML = "";
} else {
    navigator.notification.alert(
        "Use `adb` to add songs to " + app.musicPath, {}, "No Music");
}
// once you have the list of files, put the valid ones in the selector:
for (var i = 0; i < files.length; i++) {
    // if the filename is a valid file:
    if (files[i].isFile) {
        // create an option element:
        option = document.createElement("option");
        // value = song's filepath:
        option.value = files[i].fullPath;
        // label = song name:
        option.innerHTML = files[i].name;
        // select the first one and add it to the selector:
        if (i === 0) { option.selected = true; }
        songs.appendChild(option);
    }
}
app.onSongChange(); // update the current song
};

// success handler for window.resolveLocalFileSystemURI():
var foundDirectory = function(directoryEntry) {
    var directoryReader = directoryEntry.createReader();
    directoryReader.readEntries(foundFiles, failure);
};

// failure handler for window.resolveLocalFileSystemURI():
var missingDirectory = function(error) {
    navigator.notification.alert("Music directory " + app.musicPath +
        " does not exist", {}, "Music Directory");
};

// look for the music directory:
window.resolveLocalFileSystemURI(app.musicPath,
    foundDirectory, missingDirectory);
},

```

挑选歌曲后，事件监听器调用一个名为onSongChange（）的函数，该函数获取到新歌uri，然后把uri作为参数调用setSong（）函数。setSong（）函数从uri提取歌名信息，并准备好播放新歌的环境：

```

/*
changes the song URI and sets the new song:
*/

```

```
onSongChange: function(event) {
  var uri = songs[songs.selectedIndex].value;
  app.setSong(uri);
},
/*
  sets the song URI
*/
setSong: function(uri) {
  if (uri !== app.songUri) {
    app.stopAudio();          // stop whatever is playing
    app.songPlaying = null;   // clear the media object
    app.musicState = 0;       // clear the music state
    app.songUri = uri;        // saves the URI of the song
    // uses the filename for a title
    app.songTitle = uri.substring(uri.lastIndexOf('/')+1);
    $(songs).val(uri);      // ensure the UI matches selection
  }
},
```

在后面当你从NFC标签读取一首新歌时，你会看到直接调用 `setSong ()`，而不通过 `onSongChange ()`。

播放/暂停控制是通过 `toggleAudio ()` 函数实现的，它检查媒体控制器的状态，要么开始播放，要么暂停或恢复：

```
/*
  toggles audio playback depending on current state of playback.
*/
toggleAudio: function(event) {
  switch(app.musicState) {
    case undefined:           // if playback is undefined
    case Media.MEDIA_NONE:    // or if no media is playing
      app.startAudio();       // start playback
      break;
    case Media.MEDIA_STARTING: // if media is starting
      state = "music starting"; // no real change
      break;
    case Media.MEDIA_RUNNING: // if playback is running
      app.pauseAudio();        // pause it
      break;
    case Media.MEDIA_PAUSED:  // if playback is paused
    case Media.MEDIA_STOPPED: // or stopped
      app.playAudio();         // resume playback
      break;
```

```
},
```

点击播放按钮时调用startAudio () 函数，当你读取有效歌曲URI的标签时也将需要调用这个函数。如果当前没在播放，这个函数会检查是否有一首歌曲的名字，如果有，则开始播放它。

正如前面看到的，媒体API使用三个回调函数：success、error和status。它们都显示在这里。它们没做太多，除了状态处理程序，它会更新播放状态：

```
/*
 Start playing audio from your device
*/
startAudio: function() {
    var success = false;

    // attempt to instantiate a song:
    if (app.songPlaying === null) {
        // Create Media object from songUri
        if (app.songUri) {
            app.songPlaying = new Media(
                app.songUri,    // filepath of song to play
                app.audioSuccess, // success callback
                app.audioError, // error callback
                app.audioStatus // update the status callback
            );
        } else {
            navigator.notification.alert("Pick a song!");
        }
    }
    // play the song:
    app.playAudio();
},
/*
    called when playback successfully initiated.
*/
audioSuccess: function() {
    console.log("Audio success");
},
/*
    displays an error if there's a problem with playback.
*/
audioError: function(error) {
```

```
// Without timeout message is overwritten by "Currently Playing: ..."
setTimeout(function() {
    app.display("Unable to play song.");
}, 300);
},
/*
    updates the running audio status.
*/
audioStatus: function(status) {
    app.musicState = status;
},
```

播放、暂停和停止处理程序都十分相似。它们先检查看看目前有无媒体在播放，然后进行相应动作，更新按钮，在message div中显示状态和歌曲名称。代码如下：

```
/*
    resumes audio playback and changes state of the play button.
*/
playAudio: function() {
    if (app.songPlaying) {          // if there's current playback
        app.songPlaying.play();    // play
        playButton.innerHTML = "Pause"; // update the play/pause button

        // clear the message div and display song name and status:
        app.clear();
        app.display("Currently playing: " + app.songTitle);
    }
},

/*
    pauses audio playback and changes state of the play button.
*/
pauseAudio: function() {
    if (app.songPlaying) {          // if there's current playback
        app.songPlaying.pause();   // pause
        playButton.innerHTML = "Play"; // update the play/pause button

        // clear the message div and display song name and status:
        app.clear();
        app.display("Paused playing: " + app.songTitle);
    }
},

/*
```

```
stops audio playback and changes state of the play button.  
*/  
stopAudio: function() {  
    if (app.songPlaying) {          // if there's current playback  
        app.songPlaying.stop();    // stop  
        playButton.innerHTML = "Play"; // update the play/pause button  
  
        // clear the message div and display song name and status:  
        app.clear();  
        app.display("Stopped playing: " + app.songTitle);  
    }  
},
```

NFC事件处理程序

Hub通信、音乐、UI操作等功能，都可写入标签，然后让标签来触发这些功能。首先是readTag（），它会读取标签，提取NDEF消息中的所有NDEF记录，并试图解析它们。如果有一条URI记录，它会验证该URI是否是一首歌曲的地址，并尝试播放。如果是类型为text/hue的MIME记录，它会假设该记录包含有灯光的JSON编码设置信息，并尝试把这些设置信息发送到Hub：

```
/*
  reads an NDEF-formatted tag.
*/
readTag: function(thisTag) {
  var message = thisTag.ndefMessage,
    record,
    recordType,
    content;
  for (var thisRecord in message) {
    // get the next record in the message array:
    record = message[thisRecord];
    // parse the record:
    recordType = nfc.bytesToString(record.type);
    // if you've got a URI, use it to start a song:
    if (recordType === nfc.bytesToString(ndef.RTD_URI)) {
      content = ndef.uriHelper.decodePayload(record.payload);
      // make sure the song exists
      window.resolveLocalFileSystemURI(
        content,
        function() {
          app.setSong(content);
          app.startAudio();
        },
        function() {
          navigator.notification.alert("Can't find " + content,
            {}, "Missing Song");
        }
      );
    }
    // if you've got a hue JSON object, set the lights:
    if (recordType === 'text/hue') {
```

```
// tag should be TNF_MIME_MEDIA with a type 'text/hue'  
// assume you get a JSON object as the payload  
// JSON object should have valid settings info for the hue  
// http://developers.meethue.com/1_lightsapi.html  
content = nfc.bytesToString(record.payload);  
content = JSON.parse(content); // convert to JSON object  
app.setAllLights(content.lights); // use it to set lights  
}  
}  
},
```

在写模式下，你需要新建一个具有两条NDEF记录的NDEF消息，其中一条记录用于歌曲，一条记录用于灯控。灯控应该是第一条记录，所以在读模式下时，它也可以触发MIME类型监听器。由于Hub对象还包含一些不属于Hue协议的属性，所以你需要提取出lights属性，建立一个新的对象放置这些属性。第二条记录就是歌曲的名字。

一旦得到了完整的NDEF记录，你就可以使用writeTag（）函数将消息写到标签上。处理完这些后，就已经到了应用程序的结尾！

```
/*  
 * makes an NDEF message and calls writeTag() to write it to a tag:  
 */  
makeMessage: function() {  
    var message = [];  
    // put the record in the message array:  
    if (hub.lights !== {}) {  
        var huePayload = JSON.stringify({ "lights": hub.lights});  
        var lightRecord = ndef.mimeMediaRecord(app.mimeType, huePayload);  
        message.push(lightRecord);  
    }  
    if (app.songUri !== null) {  
        var songRecord = ndef.uriRecord(app.songUri);  
        message.push(songRecord);  
    }  
    //write the message:  
    app.writeTag(message);  
},  
/*  
 * writes NDEF message @message to a tag:  
 */  
writeTag: function(message) {  
    // write the record to the tag:  
    nfc.write(
```

```
message,           // write the record itself to the tag
function () {      // when complete, run this callback function:
    app.clear();   // clear the message div
    app.display("Wrote data to tag."); // notify user in message div
    navigator.notification.vibrate(100); // vibrate the device as well
},
function (reason) { // runs if the write command fails
    navigator.notification.alert(reason,
        function() {}, "There was a problem");
}
);
}
}; // end of app
```

同样，完整的应用程序源代码可以在GitHub (<http://bit.ly/moodsetter>) 上找到。

当你运行应用程序时，它会先检查是否注册到Hue Hub。如果没有注册，则会提示你按Hub上的“link”按钮。一旦连接到Hub，它就会填充Light Location下拉菜单。你将能够相互独立地改变灯光和音乐的设置，也能独立地改变模式。如果处于写模式，把一个NDEF标签放在有效范围内，将自动把当前设置信息写入到该标签。在读模式下，应用程序会自动读取标签里的歌曲和灯光设置信息。

你可能会发现，你需要保持标签和手机紧靠着一秒钟左右，以获取标签上的所有设置信息。未读完时拿开手机会造成读取错误。



如果你决定要花哨的应用程序，并添加太多的CSS，则可能会影响交互性能有一些影响。我们注意到，有些媒体API受用户界面的CSS影响比较大。解决这些问题已经超出了本书的范围。想了解更多信息，请查阅相关硬件加速CSS和PhoneGap的文档。

启用后台调度

虽然应用程序在前台运行时读取标签更直观，但是你希望它在后台运行时也能读取text/hue的MIME类型标签记录。打开AndroidManifest.xml文件，并添加一个新的intent filter，如下所示：

```
<intent-filter>
    <action android:name="android.nfc.action.NDEF_DISCOVERED"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <data android:mimeType="text/hue" />
</intent-filter>
```

更新完应用程序后，退出，试着轻触一下有text/hue记录的标签，该应用程序会自动启动，改变灯光并播放音乐。

总结

正如你所见，编写一个具有NFC功能的应用程序的大多数工作并不真的与NFC相关。真正与NFC相关的工作是思考采用什么格式的记录类型，以及如何使应用程序功能兼容各类标签。在这个例子中，你选择了一个Well-Known类型的URI记录，因为它会让调用媒体API变得很简单。还选择了一个MIME类型的text/hue记录，因为它可以轻松地调用Hue API。一旦用户界面的工作完成，整合标签读写功能的工作和之前做的例子类似，并没啥不同之处。把NFC功能封装起来使用，并提前考虑所需使用的协议和数据结构，将使编写NFC功能的应用程序更容易。

第7章 Arduino和NFC介绍

正如你在第6章中看到的，你可以通过NFC无线电连接到带有物理接口的设备上，并启动各种动作。虽然我们在之前的章节中依赖这些设备（这里主要指手机设备），但没必要一定这么做。想象一下，如果你希望Mood Setter程序可以做比控制灯光和音乐还要多的事情，则可能想要它能开或者关窗帘、锁门，甚至是放烟花。所有的这些用一个客户端控制器、一些电子元件和机械部件以及NFC无线电都是有可能实现的。智能手机和平板都不是唯一能使用NFC的设备。在市场上同样有很多支持NFC的微型控制器。本章将会讲解在Arduino微型控制器平台上的NFC使用。

数字化遇上物理：Arduino

如果你对在电子设备上创建物理接口感兴趣，那么一般都开始于微型控制器。微型控制器几乎是所有电子设备的核心地带。微型控制器的数字输入和输出都能读取来自传感器的电量变化，或者控制一些其他部件，比如电机、灯、扬声器等。Arduino微型控制器平台可以让你很容易地通过结合一些软件方面的简单集成开发环境和一系列预装单片机的电路板开发自己的微控制器项目。

常见的Arduino项目包括定制乐器、遥控宠物喂食、环境检测项目，以及提供不同形式的游戏控制器等。

Arduino的软件和硬件都是开源的，你可以在Arduino网站 (<http://www.arduino.cc>) 上找到IDE的源代码、固体件的源代码，以及Arduino的电器原理图和电路版布局规划。

Arduino是一个通用的微型控制器，它的输入和输出并不是专门针对某种特定的应用程序。你可以将其想象成手机的CPU：虽然它是可编程的，但是如果缺少蜂窝无线、屏幕控制器以及音频控制器（不是说NFC控制器），它也是没用的。为了让Arduino能发挥它应有的作用，你需要将它连接到特定的组件，比如无线电通讯、电机控制器、物理传感器等。Arduino平台采用规范的称作shield的附加模块来执行各种功能。在本章中，你需要连接一个Arduino Uno（最常见的Arduino模块）到NFC无线shield。

虽然微型控制器通常用来做电子产品开发，但是在本书中你不会得到太多关于这方面的东西。对于这里提到的项目，你将使用大部分预装好的能被Arduino或其他微型控制器控制的电子模块，你不用设计或者构建自己的电路。你可以使用shield和其他预建模块，这样你就能专注于在Arduino环境中使用NFC。

微型控制器和操作系统

Arduino的核心处理器是一个典型的非常低成本的控制器，从你的咖啡机到闹钟到热水器。即使是感应灯在电路板上都有一个微型控制器。批发Arduino的处理器大概也就1美元的成本。微型控制器的成本主要集中在支持电路、物理接口，以及使其更加容易使用的工程上。

因为这些控制器通常用于单任务应用程序，并且处理能力非常有限，所以它们通常没有操作系统的运行开销。在本书中使用的Arduino只能一次运行一个程序，而该程序会一直运行，直到微型控制器被拔掉。直到现在，默认的单片机编程还是没有操作系统，所以学习如何编写只将一件事做好的程序，是目前微型控制器编程的关键技能。

处理器的成本越来越低，支持操作系统功能的处理器成本与最有能力的微型控制器并没有多大的不同。因为Arduino的价格，Raspberry Pi开发板在微型控制器爱好者中已经掀起了一股潮流，它能运行Linux操作系统有限的版本。在不久的将来你能看到越来越多的低于50美元的嵌入式Linux开发环境。你会看到更多的在第9章中提到的类似于Raspberry Pi和BeagleBone Black的嵌入式系统。

即使成本不再是问题，操作系统也还是很容易因为应用程序的使用而造成大量的系统开销，然而很多传感器和执行器在与物理世界交互时需要处理器比传统的操作系统更迅速地做出响应。实时操作系统（RTOS）具有一些常规操作系统的能力，比如多线程，同时还保持了无系统的操作速度。一个RTOS通常缺少像文件系统或者复杂的内存管理的功能，而且也缺少像Arduino系统这样的简单概念。这就是了解嵌入式操作系统方式和单一程序方式的价值。在展示了如何在操作系统方面使用NFC之后，我们将在本章告诉你如何使用没有操作系统的方式。

NFC的硬件核心

NFC通信的核心就是无线控制器。当你读取一个（前面章节）已经用过的被动式标签时，控制器就是发起者。它能发起标签接收的无线电信号，并等待返回，就像在第2章中提到的一样。当与类似于智能手机或者平板这种活动目标通信时，无线电则可以是发起者或者作为目标。如果是目标，它就会监听无线电信号并返回。控制器就像是一个调制解调器：它控制无线电信号的发送和接收，以及将信号转换为数据与它的中央处理器进行数据交互。

NXP PN532现在是市场上最流行的NFC控制器之一，它类似于不同类型的NFC模块，并且PN53x系列的控制器在市场上的大部分具有NFC功能的电子设备中都能找到。这里有一些模块与使用PN532的Arduino兼容：Adafruit的PN532 NFC/RFID控制器接口板v1.3、Adafruit的PN532 NFC/RFID Controller Shield for Arduino，以及Seeed Studio的NFC Shield、NFC Shield v2.0。本章使用的所有这些模块都能与NDEF库兼容。

在你的Android设备中，NFC控制器与CPU的I/O引脚连接，并使用与这里类似的协议与其通信。PhoneGap的NFC插件与Arduino库有同样的作用：它允许Android设备的中央处理器来控制设备上的NFC芯片。

Arduino开发环境

为了做好Arduino的介绍练习，你需要：

- 一个Arduino Uno微型控制器，很多网店都有卖，包括Arduino、Adafruit、Seeed Studio等
- Arduino IDE (<http://bit.ly/arduino-software>)



本章的库和示例在Uno上都经过了全面的测试，但没有在Due或者Leonardo上测试，它们应该也能在使用Atmega328处理器的Arduino衍生设备上使用。对于其他的处理器，在未知领域它们可能会奏效，但也不一定。

一旦你已经下载并安装了Arduino IDE，就打开它，你会得到一个包含下列代码的编辑器窗口：

```
void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:
}
```

Arduino的编程框架是用C++写的，如果你也想为Arduino编写代码，最好也使用C++。框架已经简化了更复杂的语言问题，对于每个Arduino程序有两个主要的函数：setup（），当处理器加电或者复位时运行一次；loop（），则在setup（）调用后重复运行直到处理器断电或者复位。没有回调函数，也没有多线程，只有一个简单的循环。你可以从循环中调用函数，后面会介绍更多。

下面是你第一次运行Arduino的程序：

```
const int led = 13;      // give the LED pin number a name

void setup() {
    pinMode(led, OUTPUT); // initialize pin 13 as an output
}

void loop() {
    digitalWrite(led, HIGH); // turn the LED on (HIGH is the voltage level)
    delay(1000);           // wait for a second
    digitalWrite(led, LOW); // turn the LED off by making the voltage LOW
    delay(1000);           // wait for a second
}
```

在做完这些之后，点击下拉菜单“工具（Tools）”→“开发板（Board）”，然后选择“Arduino Uno”。在下拉菜单“工具（Tools）”→“串行端口（Serial Port）”或者是“工具（Tools）”→“Arduino更高版本端口（Port in later versions of Arduino）”中，你会看到串行端口列表。遗憾的是，它们没有一个被命名为“Arduino”，但是可以在这个列表中找到你自己的端口名字，只要将Arduino Uno插进USB接口（在Windows上你会被要求安装驱动；它们被包含在IDE的下载中），然后再重新看一下这个列表，列表中的新端口就是你的Arduino Uno。在OS X上，你会看到两个端口，其中一个标记为/dev/cu.usbmodem-xxx，另一个标记为/dev/tty.usbmodem-xxx，xxx为主板的序列号。无论哪一个起作用都没关系，因为它们是功能相同的端口。例如，在2011款的MacBook Air上运行OS X 10.8，这个端口名为/dev/tty.usbmodem621或者/dev/tty.usbmodem421。在Windows 7或者8机器上是COM3，在Linux上通常是/dev/ttym0。

选择端口，然后点击工具栏上的“上传（Upload）”按钮，显示如图7-1所示。

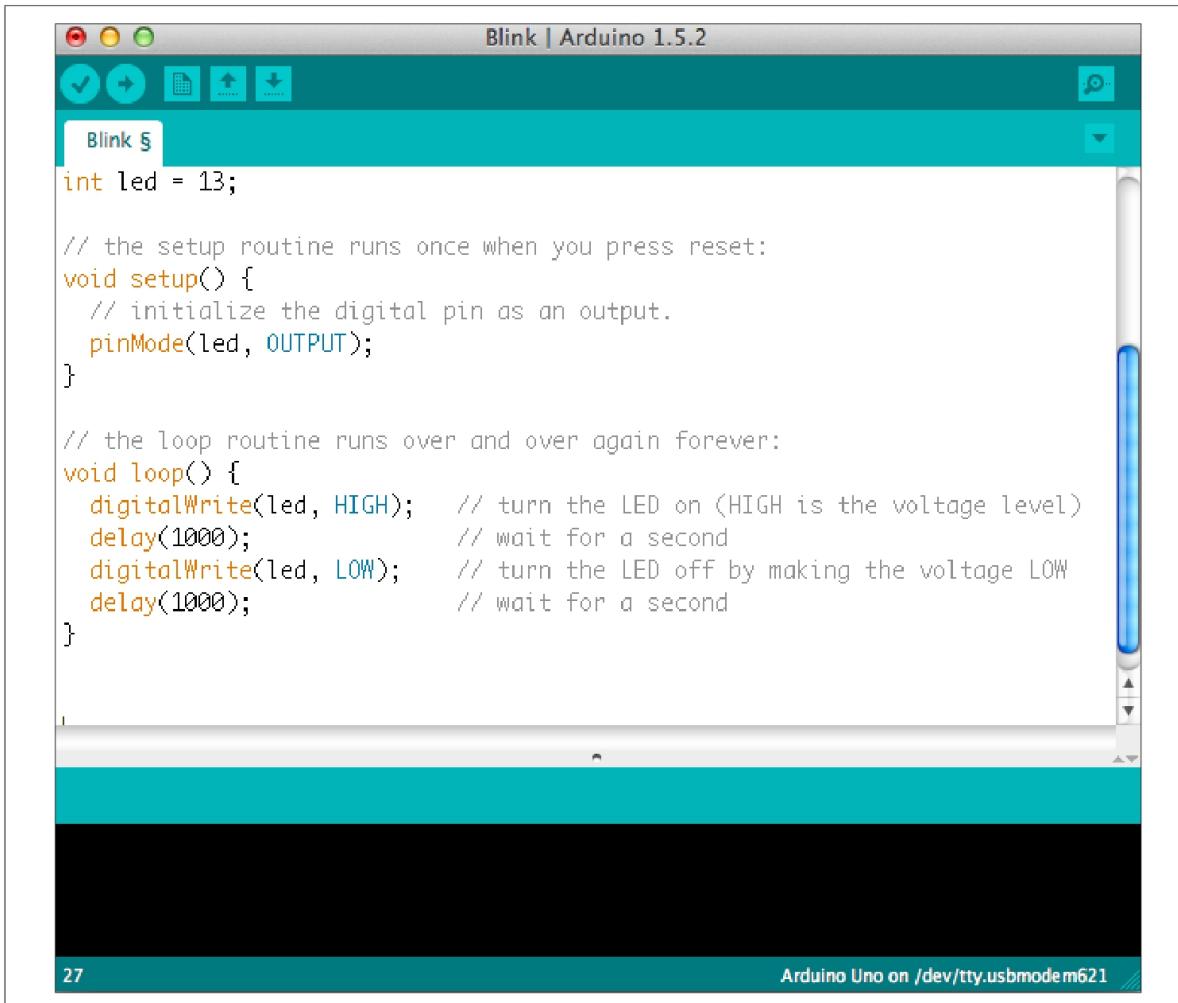


图7-1：Arduino IDE

图7-1同时还显示了工具栏按钮；从左开始，分别是验证（Verify，编译代码）、上传（Upload，编译和上传）。新建（New）、打开（Open）以及保存（Save）。最右边是串行端口监视器。版型选择和串行端口则在窗口的右下角。IDE将编译代码并上传到开发板，开发板将重置并启动运行，你会在开发板上Arduino logo的旁边看到一个闪烁的LED（见图7-2）。这个LED就是连接控制器的I/O引脚之一，数字引脚13。

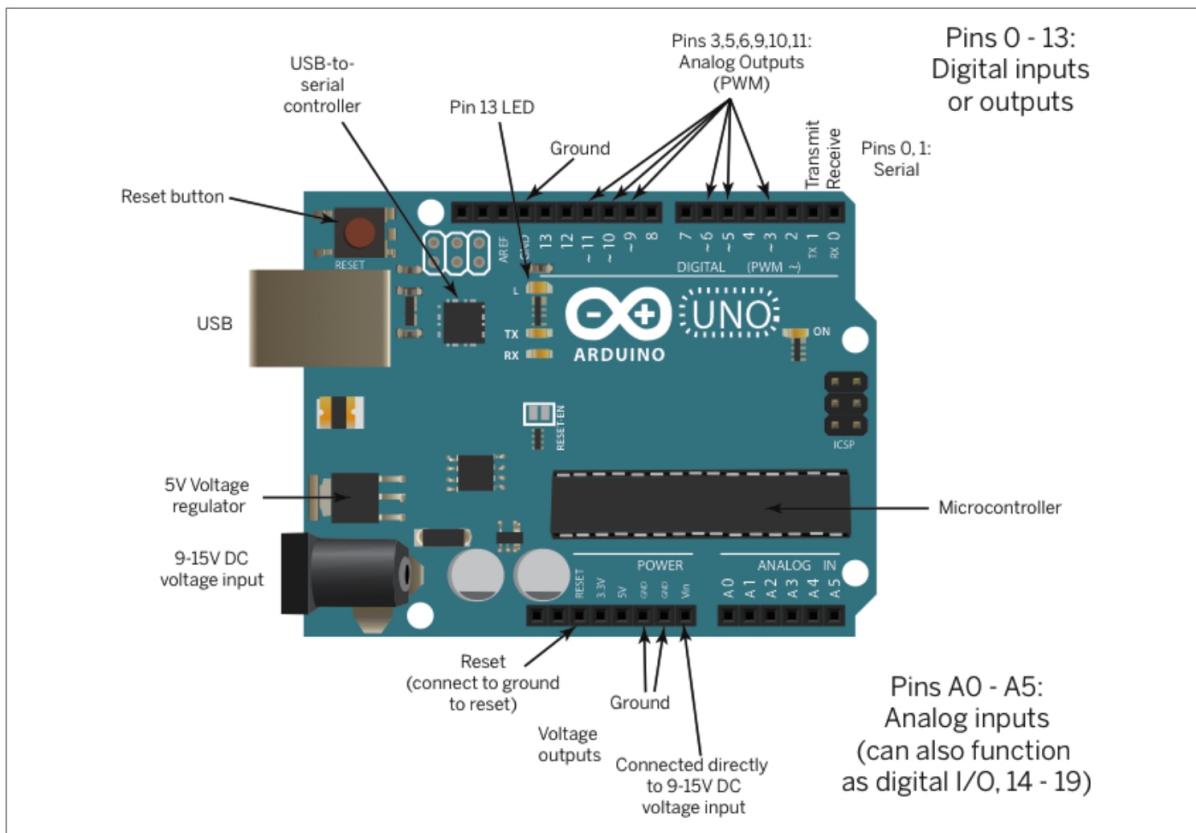


图7-2：Arduino Uno（图片来自Maker Media，改编自Tom Igoe的Making Things Talk，2nd ed.，最初由Jody Culkin绘制）

Arduino跟所有的微型控制器一样，通过I/O引脚来读取和控制电压。`digitalRead()`命令从指定的引脚返回一个1或0，也可以叫HIGH或LOW（参照电压等级）。`digitalWrite()`命令则是设置一个HIGH或者LOW。Uno工作需要5V，那么HIGH或者1就对应5V，LOW或者0则对应0V。一些I/O引脚也能在模拟模式下运行：模拟引脚A0到A5可以读取模拟电压0~5V，约0.05V电压（返回一个范围0~1023）。数字引脚3, 5, 6, 9, 10和11可以“假”连续通过脉冲针256步骤的占空比变化模拟输出电压。

在上面的代码中，Arduino的初始化数字引脚13为输出引脚，这意味着它将在那个引脚上驱动电压。在循环中，处理器设置电压到5V (`digitalWrite(ledPin, HIGH)`)，然后到0V (`digitalWrite(ledPin, LOW)`)，在这两者之间，它什么都不做，延迟1000ms (`delay(1000)`)，因为这是运行在Arduino处理器

上的唯一程序。`delay()` 的真正含义是处理器不执行任何操作，所以你不会看到它到处都在用。

IDE将在你的文档目录下为你的sketch（剪影）建立一个名为Arduino的默认目录，为了方便起见，将其保存在名为Bilink的目录下。以后你还会看到更多的sketch目录。

核心的Arduino命令参考可以在Arduino的Language Reference页面 (<http://arduino.cc/en/Reference>) 上找到，而且在Learning部分也有几十个关于如何使用该平台的例子。你还可以从“文件 (File)”菜单中选择“示例 (Examples)”，从而在IDE中找到这些例子。

串行通信

除了基本的I/O功能，Arduino也可以通过USB采用异步串行，以及I2C和PN532芯片使用的SPI协议进行通信。你会看到使用串行USB大量发送信息到电脑或从电脑接收信息，同时也用来将你编译好的程序（在Arduino中叫sketch（剪影））上传到Uno。这里有一个剪影，通过USB串口从电脑中读取数据，并使用输入来控制LED在引脚13上的输出。

```
const int led = 13;      // give the LED pin number a name

void setup() {
  Serial.begin(9600);    // open serial communication at 9600 bps
  pinMode(led, OUTPUT); // make the LED pin an output
}

void loop() {
  if (Serial.available()) {
    char input = Serial.read();           // read a byte from the serial port
    if (input == 'H' || input == 'h') {    // if it's H or h
      digitalWrite(led, HIGH);          // turn the LED on
      Serial.println(input);           // echo what the user typed
    }
    else if (input == 'L' || input == 'l') { // if it's L or l
      digitalWrite(led, LOW);           // turn the LED off
      Serial.println(input);           // echo what the user typed
    }
  }
}
```

完整的源代码可以在GitHub (<http://bit.ly/serialled>) 上找到。

上传这个sketch（剪影），然后点击工具栏右侧的“串行端口监视器（Serial Port Monitor）”，打开串行端口监视器。输入“H”或“h”，然后点击“发送”或按Enter键，如果它连接到了引脚13，那么指示灯将亮起。输入“L”或“l”，则会关闭。

一旦你理解了数字输入和输出、模拟输入和输出，以及串行通信，你就已经了解如何与Arduino一起工作的基本知识了。所有的这些要么通过读取输入引脚上的电压，在输出引脚上产生电压进行控制，要么使用串行协议发送数字交互。串行协议本身只是一系列特定I/O引脚上的电压脉冲。

更多关于开发Arduino项目的信息，请查阅Massimo Banzi的Getting Started with Arduino、Tom Igoe的Making Things Talk或者Michael Margolis的Arduino Bookbook等。你能在网上找到成千上万的Arduino例子项目、示例代码、库以及附加模块。

SPI, I2C, UART?

PN532 NFC芯片在提到的shield核心中提供了三种不同的通信接口与控制它的微型控制器通信。它能使用通用异步收发器(UART)进行异步串行通信，也可以使用两种形式的同步串行通信、串行外设接口(SPI)或内部集成电路通信(I2C)。异步串行通信的两台计算机，每台都有自己的内部时钟，并且可以独立操作，而SPI和I2C用于主计算机和依赖于主计算机设备之间的通信。异步串行始终是一对一的通信，但是I2C和SPI是总线协议，这意味着可以有多个可单独寻址的设备共享相同的通信线路。大多数可编程的微型控制器可以提供这三个接口，它们之间进一步的差异与本书无关。

不是所有的模块和这里提到的Arduino shield都提供这三个接口。Adafruit的电路板提供了三个接口，但是Adafruit的shield只提供SPI和I2C，而Seeed Studio则只提供SPI。

在撰写本文的时候，有很多Arduino库可以用于PN532控制器，其中有些使用SPI接口，有些使用I2C接口。这里使用的NDEF库是对Seeed Studio的PN532库的包装，它提供了SPI或I2C的功能。在本章中你将看到如何包含PN532库和NDEF库。

安装Arduino库

Arduino环境可以通过库进行扩展，就像任何编程平台一样。为了使用Arduino和NFC，你需要几个库。从Don的GitHub版本库（<https://github.com/don/NDEF>）中下载NDEF库。为了将其安装成一个Arduino库，在“Sketch”菜单中点击“导入库（Import Library...）”，然后选择“添加库（Add Library...）”，这样就能从ZIP文件中安装这个库了，或者选择一个目录并选择“所有文件（All Files）”。在这两种情况下，IDE将从ZIP文件或目录中导入所有文件。在每个库目录下都有一个名为“示例（Examples）”的子目录。在导入库之后，你可以在“文件（File）”菜单中通过库名称找到对应的子菜单。

你还需要一个shield适配器库。这里有几种不同类型的PN532芯片库，但是在下面例子中，你将使用Seeed Studio的PN532库，它们能同时作用于Seeed的shield和Adafruit的shield。你可以从Seeed的GitHub版本库（<http://bit.ly/seeed-github>）中下载。Seeed Studio的NFC库都被分离在三个不同的组件中：PN532文件夹、PN532_I2C文件夹和PN532_SPI文件夹。为了让它们能适配Arduino 1.5以上的版本，你需要将它们全部导入。这样，当你编译代码时就能减少内存消耗，因为你不需要加入交互协议驱动。



其他库与库命名空间冲突

Adafruit的NCFShield I2C也是一个与PN532合用很好的库。你可以从GitHub（http://bit.ly/nfcshield_i2c）上下载它。Adafruit库只能使用I2C接口，因此它也只能支持Adafruit shield，它是一个非常稳定的库，并且比Seeed库需要更少的内存。Adafruit库不能自动与Arduino NDEF库合用，但是可以稍作修改之后这样做。

如果你之前已经尝试了一些较低级别的NFC库，在你的库文件夹中仍然有早期的版本（就像我们，之前做的一样），那么当你编译代码的时候，可能会出错。如果出错了，就从你正在使用的Arduino库目

录下删除与PN532相关的库并重启IDE。你可以在你的Arduino sketch（剪影）目录下找到库目录。

对于这些示例，你需要：

- 一个Arduino Uno微型控制器
- 一个Arduino IDE
- 一个NFC Shield（你可以使用Adafruit的PN532 NFC/RFID Controller Shield (<http://bit.ly/pn532>) 或者Seeed Studio的NFC Shield (<http://bit.ly/seeed-sld80453p>)、NFC Shield v2.0 (<http://bit.ly/seeed-nfc-shield>)）
- 如果你正在使用Adafruit shield，你可能也会希望从Adafruit得到一些盾叠加头（Shield Stacking header）(<https://www.adafruit.com/products/85>)，为了使用这个shield，你不得不做一个焊接，但是它是值得的
- 一些NFC标签（在本章中，Mifare Classic与你正在使用的Arduino库是最兼容的，尽管它们在一些Android设备上无法正常工作；更多的关于标签与设备工作的内容详见“设备与标签类型匹配”一节）
- Arduino IDE 1.5.3或更高版本 (<http://bit.ly/arduino-software>)
- Arduino的NDEF库，可以从Don Coleman的GitHub版本库 (<https://github.com/don/NDEF>) 中找到
- SeeedStudio PN532库 (<http://bit.ly/seeed-github>)

Adafruit shield原装是没有脚头（pin header）安装的，所以你需要将这些焊接上，可能会很麻烦，但是Adafruit有一个很好的教程，通过上面的图片你可以知道如何将shield插入Arduinio Uno（见图7-3）。

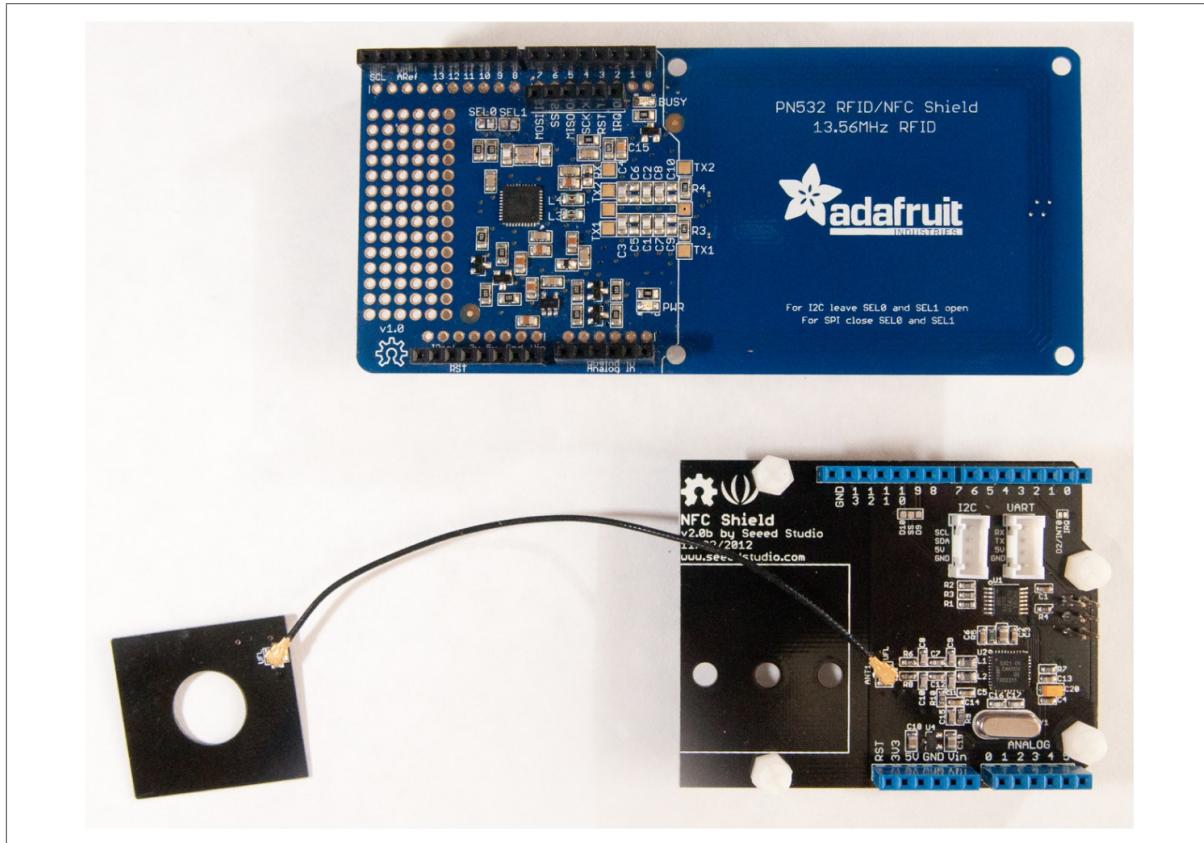


图7-3: Adafruit NFC Shield (上图) , Seeed Studio NFC Shield 2.0 (下图) 。

Arduino的NDEF库

你正在使用的库可以被看作一个堆栈，PN532适配器库提供了低级别命令在shield中与PN532芯片交流，一个使用SPI，另一个使用I2C。它们只是从标签中传递字节字符串。为了解析这些字节，你需要一些辅助库来描述各种标签协议：Mifare Classic、Mifare Ultralight等（在第2章中有介绍）。上面的标签类型库是一个NFC层，去除标签之间的区别，并提供一个字节流。最后，在此之上，还有NDEF库，读取标签类型库的一系列NDEF消息，并传递记录的字节数。图7-4显示了堆栈和库的关系。

如果没有NDEF库，你必须弄清楚从标签来的字节到字节是怎么回事，有了它之后，你通常可以忽略底层到底是怎么回事，而专注于读取和写入NDEF消息。

并不是所有的标签类型辅助库都能被写入。到目前为止，NDEF库包含读取和写入Mifare Classic和读取兼容的类型2（Mifare Ultralight）标签库，而不是其他的，所以，如果你需要使用这两种标签类型的一种，或者准备写你自己的标签描述库。

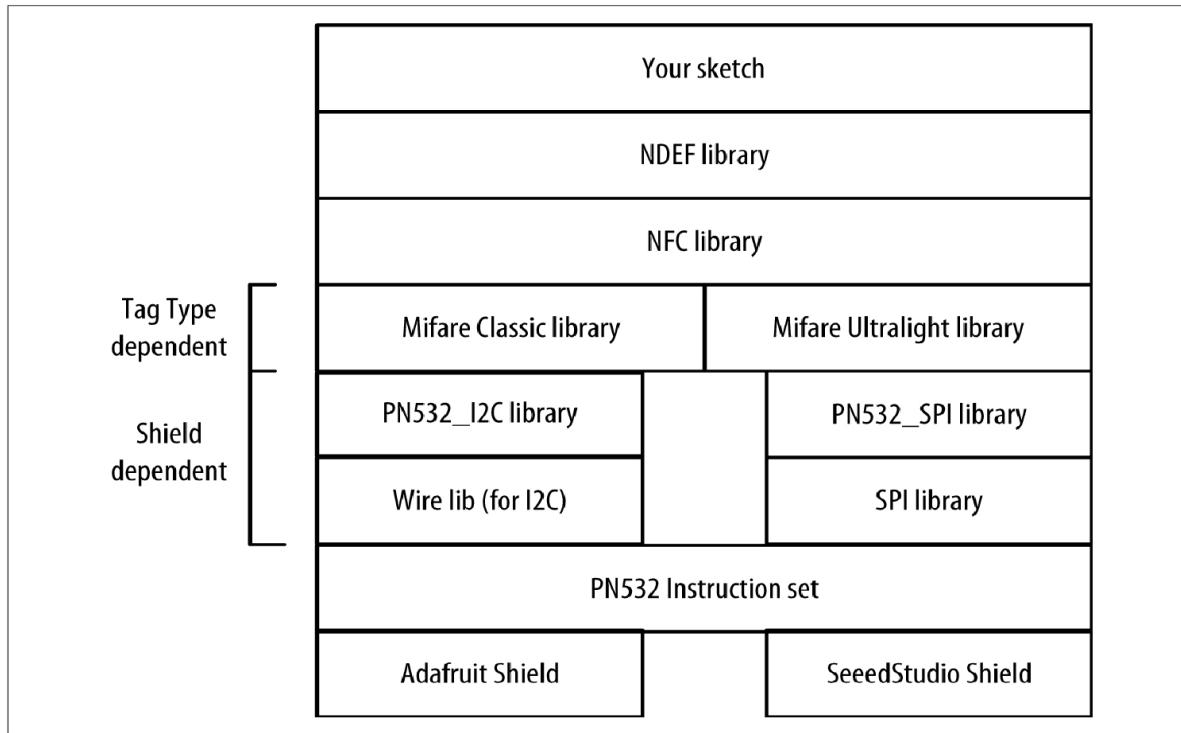


图7-4：Arduino的NFC协议栈库

Arduino NDEF库包含了一些你应该已经熟悉的对象和概念：

NfcAdapter

这个对象是低级适配器库实例，它主要用来寻找标签。

NfcTag

这个对象包含标签的元数据：UID、技术和规模。

NdefMessage

当你读到一个NDEF格式的标签时，你会得到一个NdefMessage，当然，这个对象处理NDEF格式字节流的编码和解码。

NdefRecord

这个对象携带有效载荷和一些元数据，比如语言、TNF、类型等。

这个库有一些读取和写入信息的辅助函数，你不会使用所有的这些函数，但是大多数函数似乎都基于你已经知道NDEF是什么了。所有的函数都按类列在了表7-1中。

表7-1：Arduino NDEF库函数

NfcAdapter 函数	NfcTag 函数	NdefMessage 函数	NdefRecord 函数
begin(void)	getUidLength()	getEncodedSize()	getEncodedSize()
tagPresent()	getUid()	encode()	getTypeLength()
read()	getUidString()	addRecord()	getPayloadLength()
write()	getTagType()	addMimeMediaRecord()	getIdLength()
	hasNdefMessage()	addTextRecord()	getTnf()
	getNdefMessage()	addUriRecord()	getPayload()
		addEmptyRecord()	getType()
		getRecordCount()	getId()
		getRecord()	setTnf()
			setType()
			setPayload()
			setId()

读取Arduino中的NDEF

NDEF库附带的ReadTagExtended例子展示了这个库的许多函数和工作流，你可以通过点击“文件（File）”菜单，选择“示例（Examples）”，然后选择“NDEF”子菜单，再选择“ReadTagExtended”，将它上传到Arduino并试着读取你之前写的标签。

你需要根据所使用的shield模块来修改这个sketch（剪影），如果你使用的是类似于Seeed Studio shield基于SPI的shield，那么你的代码开始段就应该像下面一样：

```
#include <SPI.h>
#include <PN532_SPI.h>
#include <PN532.h>
#include <NfcAdapter.h>

PN532_SPI pn532spi(SPI, 10);
NfcAdapter nfc = NfcAdapter(pn532spi);
```

如果你使用的是基于I2C的shield，比如Adafruit shield，那么你的代码开始段看起来就像这样：

```
#include <Wire.h>
#include <PN532_I2C.h>
#include <PN532.h>
#include <NfcAdapter.h>

PN532_I2C pn532_i2c(Wire);
NfcAdapter nfc = NfcAdapter(pn532_i2c);
```

之后，一切都是相同的，不管你用的是什么类型的shield。本章的后面部分给出的是基于I2C的shield的代码，如果你正在使用基于SPI的shield，那么你就需要改变代码的开始段。

正如你从下面代码中看到的，程序逻辑是你之前似乎见过的。首先使用nfc.tagPresent () 函数检查标签是否存在。由于Arduino没有事件处理程序，所以重复轮询是必要的。当你知道一个标签的存在后，你可以使用NfcTag tag=nfc.read () 得到一个标签对象并读取数据。接下来使用tag.getNdefMessage () 函数检查是否有消息。如果有，则遍历消息中的记录，并解析出消息属性（TNF、类型、有效载荷和ID）。

```
#include <Wire.h>
#include <PN532_I2C.h>
#include <PN532.h>
#include <NfcAdapter.h>

PN532_I2C pn532_i2c(Wire);
NfcAdapter nfc = NfcAdapter(pn532_i2c);

void setup() {
  Serial.begin(9600);
  Serial.println("NDEF Reader");
  nfc.begin();
}

void loop() {
  Serial.println("\nScan a NFC tag\n");

  if (nfc.tagPresent()) {
    NfcTag tag = nfc.read();
    Serial.println(tag.getTagType());
    Serial.print("UID:");
    Serial.println(tag.getUidString());

    if (tag.hasNdefMessage()) { // every tag won't have a message
      NdefMessage message = tag.getNdefMessage();
      Serial.print("\nThis NFC Tag contains an NDEF Message with ");
      Serial.print(message.getRecordCount());
      Serial.print(" NDEF Record");
      if (message.getRecordCount() != 1) {
        // if there's more than one record, pluralize:
        Serial.print("s");
      }

      // cycle through the records, printing some info from each
      int recordCount = message.getRecordCount();
      for (int i = 0; i < recordCount; i++)
      {

```

```

Serial.print("\nNDEF Record ");
Serial.println(i+1);
NdefRecord record = message.getRecord(i);

Serial.print(" TNF: ");
Serial.println(record.getTnf());
Serial.print(" Type: ");
Serial.println(record.getType()); // will be "" for TNF_EMPTY

// The TNF and Type should be used to determine
// how your application processes the payload
// There's no generic processing for the payload.
// it's returned as a byte[]
int payloadLength = record.getPayloadLength();
byte payload[payloadLength];
record.getPayload(payload);

// Force the data into a String:
String payloadAsString = "";
for (int c = 0; c < payloadLength; c++) {
    payloadAsString += (char)payload[c];
}
Serial.print(" Payload (as String): ");
Serial.println(payloadAsString);

// id is probably blank and will return ""
String uid = record.getId();
if (uid != "") {
    Serial.print(" ID: ");Serial.println(uid);
}
}

}

}

delay(3000); // delay before next read
}

```

上传这段代码，然后通过点击IDE窗口的右上角的图标打开串行端口监视器。当你运行前面章节中提到的标签时，你应该得到的输出看起来像这样：

```

NDEF Reader
Found chip PN532
Firmware ver. 1.6
Scan a NFC tag
Mifare Classic
UID: A4 AF 77 5D

```

This NFC Tag contains an NDEF Message with 2 NDEF Records

NDEF Record 1

TNF: 2

Type: text/hue

Payload (as String): {"lights":{"1":{"name":"Living room","state": {"on":true,"bri":223, "hue":47293,"sat":0}}}}

NDEF Record 2

TNF: 1

Type: U

Payload (as String): /sdcard/myMusic/Forever.mp3

tagPresent和delay函数都是阻断函数，这意味着当它们完成执行后就停止了该程序。只要设置延迟，你就需要设置tagPresent大约为100ms。你应该只在控制器不做任何操作的时候调用它们。如果你需要读取物理输入，比如pushbuttons或者接收串行接口的数据，则应优先考虑这些，就跟你在后面将看到的应用程序sketch（剪影）一样。

在Arduino中写入NDEF

将记录写入到一个NDEF消息中并发送其实也很简单。

WriteTagMultipleRecords例子给了你一个很好的注意。点击“文件（File）”菜单，选择“示例（Examples）”，然后选择“NDEF”子菜单，再选择“WriteTagMultipleRecords”，上传到Arduino并试着写入标签，然后用手机读取它。

首先，创建一个NdefMessage对象，然后使用写入辅助函数添加记录，比如addTextRecord（）、addUriRecord（）或者addMimeMediaRecord（）。一旦消息被组装好，你就可以使用NfcAdapter write（）函数发送它，它会返回成功或者失败。

```
#include <Wire.h>
#include <PN532_I2C.h>
#include <PN532.h>
#include <NfcAdapter.h>

PN532_I2C pn532_i2c(Wire);
NfcAdapter nfc = NfcAdapter(pn532_i2c);

void setup() {
    Serial.begin(9600);
    Serial.println("NDEF Writer");
    nfc.begin();
}

void loop() {
    Serial.println("\nPlace a formatted Mifare Classic NFC tag on the reader.");
    if (nfc.tagPresent()) {
        NdefMessage message = NdefMessage();
        message.addTextRecord("Hello, Arduino!");
        message.addUriRecord("http://arduino.cc");
        message.addTextRecord("Goodbye, Arduino!");
        boolean success = nfc.write(message);
        if (success) {
            Serial.println("Success. Try reading this tag with your phone.");
        } else {
            Serial.println("Write failed");
        }
    }
}
```

```
        }  
    }  
    delay(3000);  
}
```

一旦你了解NDEF库的基本读写功能，你就已经准备好用它去构建应用了。

微型控制器NFC应用：酒店钥匙卡

如果你在过去的的10年里入住过酒店，就有很好的机会得到一个基于RFID的房间钥匙卡。这是一个很容易通过使用NFC技术得到提升的应用，所以作为嵌入式NFC的例子，你将编写一个简单的基于NFC的酒店房间钥匙卡的应用程序。在这个应用中，你会写一个Arduino的NFC读取器sketch（剪影）以及一个NFC写入器sketch。后者将涉及额外的硬件：一个电控门锁（如果你想，你可以在真正的门上使用。如果不能让真实的物理世界发生点什么，那做这个还有什么乐趣呢）。

此应用程序的前半部分是你所写的Arduino标签写入器。你将制作一个小盒子位于酒店登记台的后面连接到台式电脑。服务员输入你的姓名、房间号，以及你住多少晚，然后在盒子里插入该卡，那么标签写入器sketch（剪影）就会将这些包含了JSON字符串的信息作为MIME媒体类型写入标签。

这个应用程序的后半部分是你所写的一个门锁控制器。Arduino将连接到门锁，当它读卡时，它会检查当前时间是否在办理入住和结账之间，如果卡上的房间号对应房门号，那么就是正确的，Arduino将会打开门闩，这样你就可以开门了。

最后，你将编写一个基于浏览器的用户界面与NFC标签写入器连接，这样前台服务员就可以在熟悉的方式下与设备进行交互。

程序的前半部分，你可以使用已经学到的方法将其移到移动设备上，第二半部分因为物理门锁，需要一个控制器，你同样需要考虑使用它的物理环境。例如，一个酒店客户通常携带了很多东西：手提箱、外套、钱包或公文包，以及房间钥匙，当她打开房间的时候，你不能指望她做多点击或者插入钥匙，你得确保在她进门后门能自动锁上。

对于这个应用程序，你需要与前面的NFC例子相同的部件。此外，你需要一些新的部件：

- 一个无焊接电路板或者Arduino的原型shield，以下是一些选项：
 - Arduino Proto Shield Rev3 (<http://bit.ly/17uFcHD>)
 - Adafruit的Arduino Proto Shield (<http://www.adafruit.com/products/51>)
 - 一个小型电路试验板 (<http://www.adafruit.com/products/65>)
(对于前两个来说，这个是可选的)
 - 如果你不需要将这些都安装在一起，则可以只使用一半大小的电路板 (<http://www.adafruit.com/products/64>)
- 一个电磁驱动的门锁，12V或更小。还有其他很多小的螺线管。我们通常使用在亚马逊买的Amico 0837L DC 12V8W开放框架式电磁电动门锁 (<http://amzn.to/1aEQe7S>)，但是你也可以从其他零售商那里买到螺线管。Adafruit销售一种类似的锁扣式螺线管 (<http://www.adafruit.com/products/1512>) 以及Seeed Studio销售几款类型，所以如果你从他们那里订购shield，同样也可以在那里买螺线管
- 一个TIP120 Darlington晶体管 (<http://www.adafruit.com/products/976>)
- 一个12V、1000mA电源，具有2.1mm内径，5.5mm外径，中心呈阳性连接器来驱动电磁阀电路 (<http://www.adafruit.com/products/798>)
- 跨接线或22AWG实心线 (<http://www.adafruit.com/products/759>)
- 两个LED（红，绿）。这些都可以从任何电子产品零售商那里买到，但仅供参考，请查阅Adafruit的红色LED组 (<http://www.adafruit.com/products/299>) 或绿色LED组 (<http://www.adafruit.com/products/298>)
- 两个220Ω电阻的LED，这也可以从任何电子产品零售商那里买到
- Michael Margolis编写的Arduino Time库 (<http://bit.ly/time-library>)

如果你计划构建和部署完整的NFC写入器和NFC读取器/锁控制器，就需要两个Arduino和两个NFC shield。不过，你只学习一个完整的应用程序就好了。

图7-5显示了应用程序的系统框图。

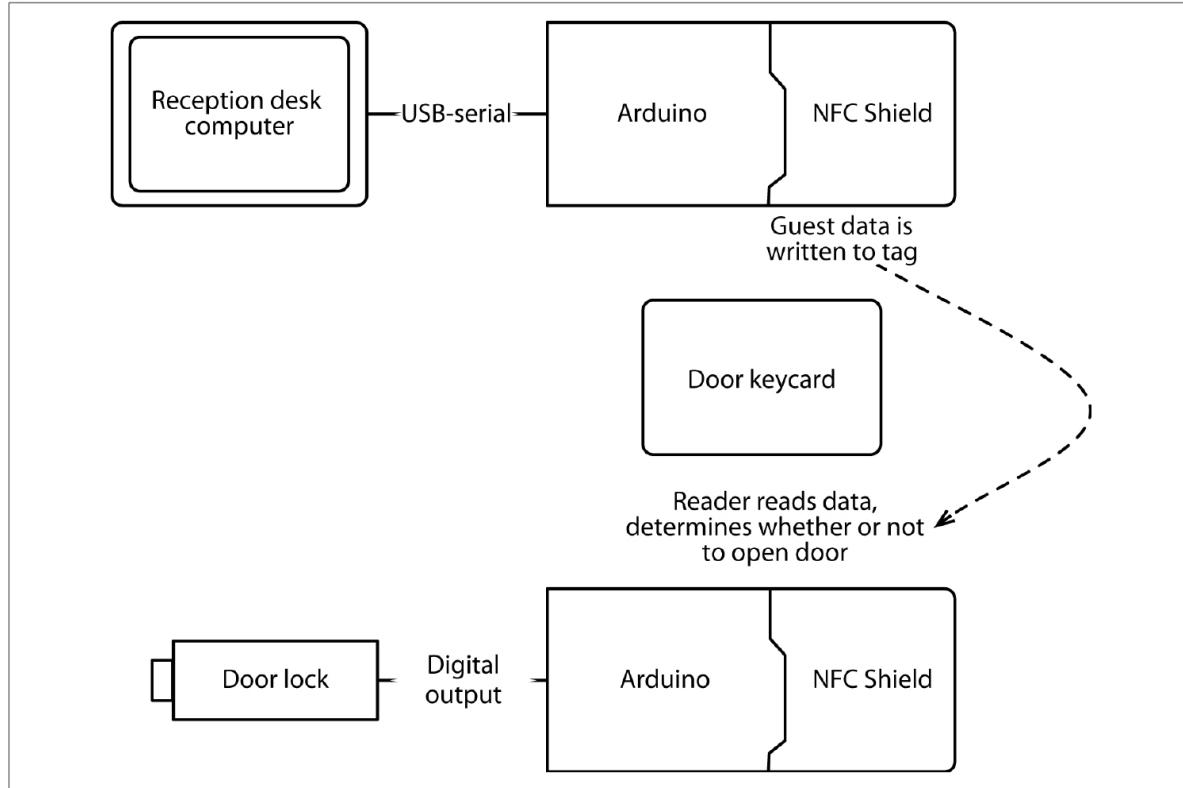


图7-5：酒店房间的钥匙卡应用程序系统框图

交互和数据格式

这个应用程序中的两个控制器，卡写入器和读取器/门锁控制都应该尽快地响应，所以最好尽可能的简单。正如你在图中看到的，写入器始终连着拥有更丰富物理接口的个人电脑。这样就能获取用户输入、时间戳，并提示用户使用电脑工作。这个微型控制器只需要等待字符串数据写入卡，并报告是否成功写入。它不关心这个数据是什么，它只是将文本数据写入。为了让本书的应用程序有更好的兼容性，我们将使用MIME媒体记录包含基本细节的JSON字符串，如下所示：

```
{  
    name: username,  
    room: (room number),  
    checkin: (checkin time in unix time),  
    checkout: (checkout time in unix time),  
}
```

写入器会等待}字符结尾的字符串序列，然后在该领域寻找标签，如果它看到一个标签，则将写入字符串（包含一条NDEF MIME媒体类型记录作为NDEF消息）。如果成功，它将打开引脚9上的绿色LED灯；如果失败，则打开引脚8上的红色LED灯。每次会检查3秒钟，然后关闭两个LED灯。

读取器则更复杂一点，因为它是一个独立的设备，不依附于其他任何电脑。这需要它本身能保持时间，读取和解析标签数据，检查当前时间来比对入住和结账时间，检查房间号和自己的编号，并提示用户是否已经读取了标签，并适当控制锁。

读取器的主要循环必须非常敏感，毕竟它的任务就是监听标签。当它没有监听标签时，应该打开一个LED指示灯，红色或者绿色，表明它当前是一个好的还是坏的读取器。如果是好的，它也应该能控制门锁。

因为你是使用JSON进行数据交换的，那么也可以按此添加其他的功能。你可以为客户写一个PhoneGap应用程序来显示房间号、入住和结账时间，你也可以将一系列的房间进入和退出时间写到卡上。这种体系结构的好处在于将其他普通的信息添加到卡上，其他的物理设备也能读取，这样在不影响锁门功能的前提下还能做其他的事情。

保持时间

UNIX和其他POSIX操作系统使用的时间秒数是从1970年1月1日开始计算的，不计算闰秒。这种格式被称为UNIX时间，更准确的说是POSIX时间，它是从1970年1月1日午夜开始标记的，被称为协调世界时（UTC）。这种时间格式可以很容易地计算时间之间的差异，并允许将时间压缩成一个单一的长整数。如果你以前没有见过它，那么现在就了解它。大多数编程语言会提供一个方法来转换UNIX时间，因此用它来传递时间是一个方便的万能方法。对于Arduino而言，你会在后面看到一个方便的库，用来保持时间。

在下列例子中你所面临的挑战之一是，不同的库和平台使用不同的尺度来表示这段时间。Date.valueOf()，是JavaScript用来返回UNIX时间的标准方式，返回的是自1970年1月1日开始的毫秒数，不同的Arduino库返回的秒数不一样。所以转换是必要的，但是它们大多数是兼容的。

如果你希望看到一些UNIX日期和时间，那么打开浏览器的JavaScript控制台并输入以下内容：

```
> var d = new Date("January 1, 2000");
> console.log(d);
> d = new Date("July 10, 1856");
> console.log(d);
> d = new Date("January 7, 1943");
> console.log(d.valueOf())
```

你会发现从1970年1月1日以后的日期都是正数，之前的都是负数。然后试试与自己相关的日期。

Arduino的NDEF写入器设备

图7-6显示的是NDEF写入器的电路。它插入Arduino，并且该电路位于shield的顶部（图中没有显示）。它包含了两个LED和NDEF shield。如图所示连接螺线管电路，你应该使用之前的Blink sketch（剪影）测试LED。将输出引脚从13改变为8或者9，然后重新上传代码。LED应该会闪烁。

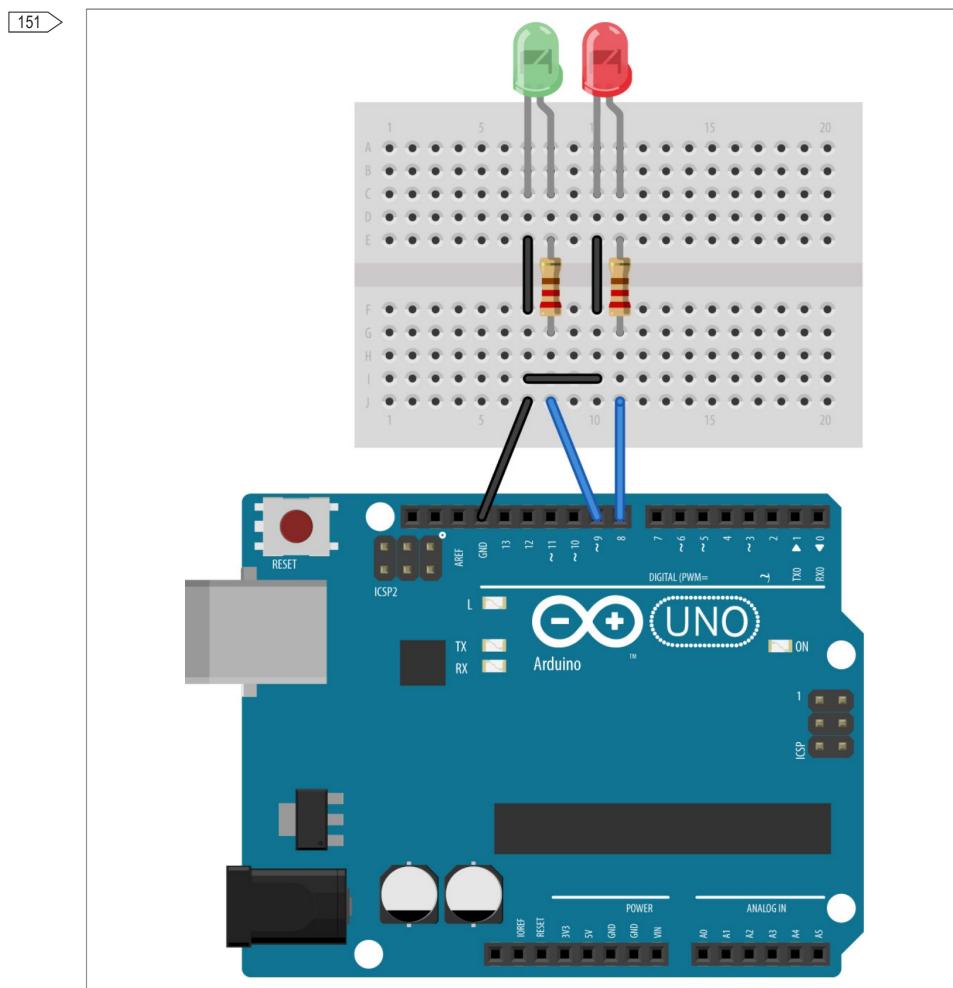


图7-6：NFC读取器门锁装置的电路

图7-7所示为NDEF写入器示意图。

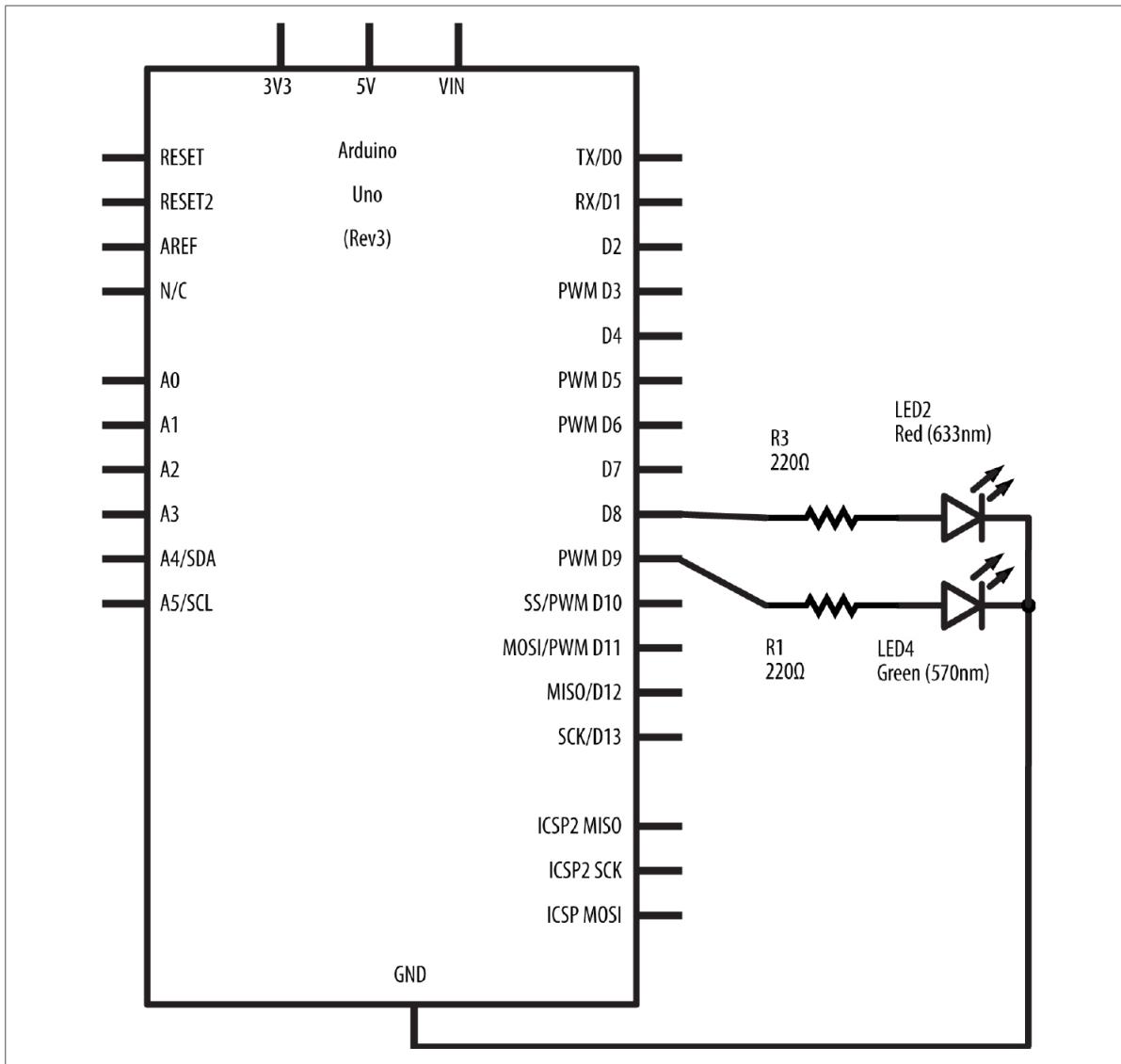


图7-7：NFC读取器门锁装置的电路示意图

从写入器代码开始，由于简单，你已经能在Android设备上运行这个程序来进行检查了，包括你需要的库，以及设置一个NfcAdapter对象。在下面的代码中你会看到一些输出引脚和串行输入的常量，以及一个跟踪LED指示灯变量。然后在setup（）函数中初始化串行通信和NfcAdapter对象，并设置两个LED指示灯引脚为输出。

```
// If you're using an SPI-based shield, change these to include the SPI library:  
#include <Wire.h>  
#include <PN532_I2C.h>  
#include <PN532.h>
```

```

#include <NfcAdapter.h>
#include <Time.h>

PN532_I2C pn532_i2c(Wire);
NfcAdapter nfc = NfcAdapter(pn532_i2c);

const int greenLed = 9;      // pin for the green LED
const int redLed = 8;        // pin for the red LED

String inputString = "";     // string for input from serial port
long lightOnTime = 0;        // last time the LEDs were turned on, in ms

boolean readyToWrite = false; // true when you are ready to write to NFC tag

void setup() {
  Serial.begin(9600);        // initialize serial communications
  nfc.begin();                // initialize NfcAdapter
  pinMode(greenLed, OUTPUT); // make pin 9 an output
  pinMode(redLed, OUTPUT);   // make pin 8 an output
}

```

该循环的主要功能就是不断地从个人电脑的桌面应用程序得到输入，当得到数据后，则调用另外一个函数去寻找标签并写入数据。以下是主循环：

```

void loop() {
  // if there's incoming data, read it into a string:
  while (Serial.available() > 0) {
    char thisChar = Serial.read();
    // add incoming character to the end of inputString:
    inputString += thisChar;
    if (thisChar == '{') {
      // new message, reset buffer
      inputString = "{";
      readyToWrite = false;
    }
    else if (thisChar == '}') {
      // end of message, ready to write to tag
      Serial.println("Ready to write data to tag");
      readyToWrite = true;
    }
  }
  // keep looking for a tag to write to when
  // you've got a string to write:
  if (readyToWrite) {
    lookForTag();
  }
}

```

```
}

if (millis() - lightOnTime > 3000 ) { // check every three seconds
    digitalWrite(greenLed, LOW);      // turn off pin 9
    digitalWrite(redLed, LOW);       // turn off pin 8
}
}
```

主循环最后的if语句使用了millis () 方法，它返回一个毫秒数，如果这个毫秒数从Arduino复位检查到打开LED成功或者失败的时间超过3秒钟，它将关闭两个LED。

当你有一个完整的字符串用来查询标签并将输入字符串写入标签时就会调用lookForTag () 函数。当有一个有效的字符串被写入时，主循环就会不停地调用lookForTag () 函数直到成功为止。如果成功写入标签，就返回true，否则返回false。

```
void lookForTag() {
    if (nfc.tagPresent()) {           // if there's a tag present
        NdefMessage message;         // make a new NDEF message
        // add the input string as a record:
        message.addMimeMediaRecord("text/hotelkey", inputString);
        boolean success = nfc.write(message); // attempt to write to the tag

        if (success) {
            // let the desktop app know you succeeded:
            Serial.println("Result: tag written.");
            digitalWrite(redLed, LOW);    // turn off the failure light if on
            digitalWrite(greenLed, HIGH);  // turn on the success light
            lightOnTime = millis();
            readyToWrite = false;        // clear write flag
        }
        else {
            // let the desktop app know you failed:
            Serial.println("Result: failed to write to tag");
            digitalWrite(greenLed, LOW);  // turn off the success light if on
            digitalWrite(redLed, HIGH);   // turn on the failure light
            lightOnTime = millis();
        }
    }
}
```

此函数也会发送一系列响应而不管写入标签是成功还是失败，两个响应都包含同一个词：Result:。这很重要，因为它代表着桌面应用程序可以很容易地解析这个响应，而不管这个标签是否被写入了数据。

完整的源代码可以在GitHub (<http://bit.ly/HotelTagWriter>) 上找到。

这就是整个sketch（剪影）。保存它并将其上传到Arduino，然后打开串行端口监视器测试，将一个标签放置在shield天线下并输入下列JSON对象（你可能希望事先输入到一个文本文件中，再剪切和粘贴，没关系，你觉得怎么样做好就怎么做）。

```
{  
  "name": "Nicky",  
  "room": 3327,  
  "checkin": 1357534800,  
  "checkout": 1373428800  
}
```

当你输入数据时，绿色LED应该会亮，而且你应该得到消息“Result: tag written”。

为了测试这些标签，打开Andriod设备上的标签读取应用程序，比如NXP TagInfo，并试着读取标签，你应该会得到相同JSON字符串的MIME媒体类型记录。

你的读取器设备跟红色和绿色LED相比，没有用户接口，它依赖于一个可以读取和写入串行协议并为前台服务员提供接口的桌面应用程序。你也不能期望服务员必须打开Arduino IDE和串行端口监视器，并输入一个JSON字符串写入卡中。相反，你要写一个桌面应用程序，这样才能从键盘输入，将其转换成JSON字符串，并发送到Arduino。

你可以在任何编程环境下通过写一个桌面接口来访问电脑的串行端口。之前你已经建立的串行通信协议将会有很大的帮助。即：

- 每条发送给写入器设备的消息都应该是一个以}结尾的JSON字符串。

- 每条从写入器设备发出的响应都应该包含“Result: ”来报告写入标签是成功还是失败。
- 每条从写入器设备发出的响应都应该结束在一个新行上（因为你使用Serial.println()）。

后面你会使用Node.js为这个设备写一个基于浏览器的用户界面，但是现在，我们把目光重新移到读取器设备上。你可以在图7-8中看到写入器设备被放在一个门上。虚线表示其中的一个控制箱和电磁阀可以被最终地嵌入到门上。而卡则是从读取器的顶部或者前面插入插槽的（大多数酒店房间都是这样的）。

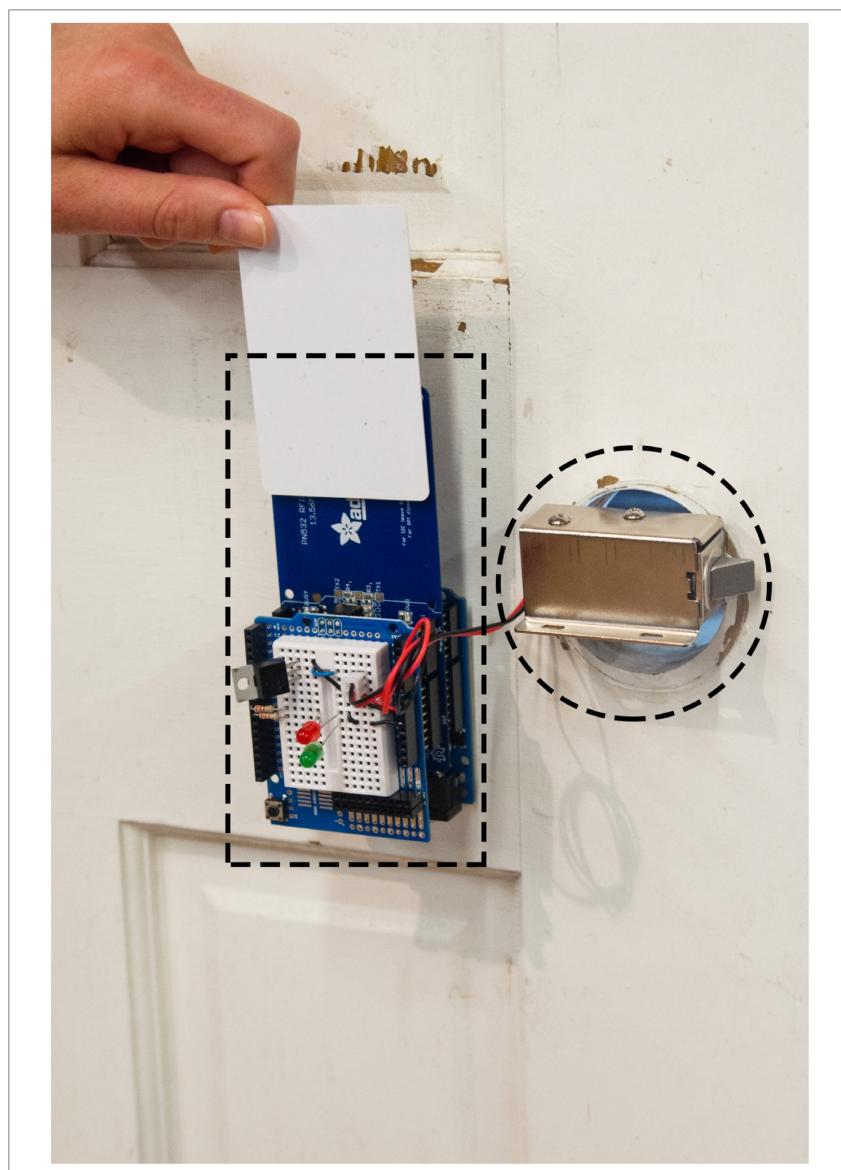


图7-8：门上的写入器设备原型

如何制作一条安全的NDEF记录

到现在为止，你可能在想：既然任何读了这本书的人都能很容易地通过重写标签进入我的酒店房间，那我为什么还要使用NDEF酒店钥匙呢？这是个好问题。你可以以你希望的任何方式对内容加密，只要读取器的控制器知道如何解密就行了。如果你想提供身份验证，NFC论坛定义的签名记录类型定义（Signature Record Type Definition）可以处理这样的情况。版本1.0是目前采用的标准，版本2.0是候选的规范。两个都能在NFC论坛规范页面 (<http://bit.ly/nfc-tech-specs>) 上找到详细的介绍。

签名RTD（Signature RTD）为开发者使用存储的公共密钥作为NDEF消息的一部分制定了标准。使用这个标准，开发者应该在每个NDEF消息中加入一条签名记录来验证记录的真实性。签名包含一个加密的哈希值和证书链，由写入器加密，读取器解密。如果能将数据解密，那么说明这个数据是有效的，如果不行，则读取器会自动忽略数据。写入器和读取器都要共享准确的加密和解密私钥。

签名RTD规范并不要求一个特定的公钥密码基础设施，但它支持现有的各种标准。

这个规范是一个认证方案，而不是一个加密方案。除了签名记录，其他的记录仍然可以被任何其他的读取器读取，但它们不会被验证。为了更安全，你可以将应用程序设计成加密数据本身，包括对加密数据的签名，那么读取器将先验证签名，然后再使用相同的算法加解密数据。

Arduino的NDEF读取器和门锁装置

NDEF读取器设备比写入器设备稍微复杂一点，它必须解析标签中的JSON数据并响应相应的动作。为了做到这点，它不仅要读取标签，还要解析数据。它必须知道是否是正确的房间号，当前时间是否在你住的这几天之间（时间是否合法）。为了做到这点，它必须能读取UNIX时间（Michael Margolis的Arduino Time库能帮忙做到这点）。通常你需要用真正时间表（RTC）芯片（比如Dallas Semiconductor的DS1307（<http://www.adafruit.com/products/264>）或DS3231（<http://www.adafruit.com/products/255>）的接口）来做，但是这个库也能使用Arduino内部的时间来做。后者虽然不是很准确，但是不需要额外的硬件，使用在这个例子中还是能做得很好的。

图7-9所示的是带有电磁锁的读取器电路（NFC shield没有显示）。它被插入到Arduino并且包含两个LED、NFC shield以及门锁的电磁阀部分。如图所示连接螺线管电路。需要注意的是，由于电磁阀需要12V、667mA操作，所以你需要在零件清单中提到12V电源供电。你也应该使用之前的Blink sketch（剪影）测试电磁阀；只是将输出引脚从13改变到7，然后再次上传代码。电磁阀应该会开一下。注意，这里的MOSFET晶体管可以使用TIP120 Darlington晶体管进行替换，因为它们具有相同的引出线。

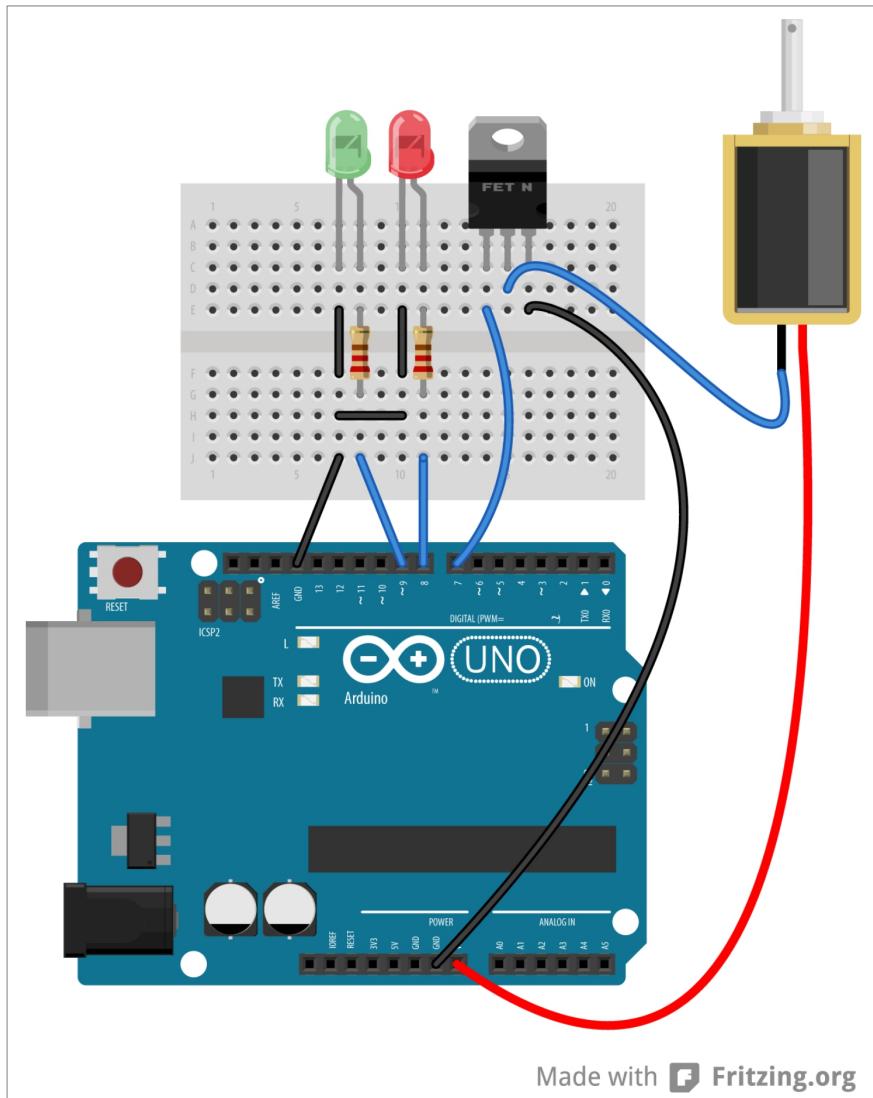


图7-9：NFC读取器设备的电路（电路板视图）

图7-10所示为NFC读取器设备的电路示意图。

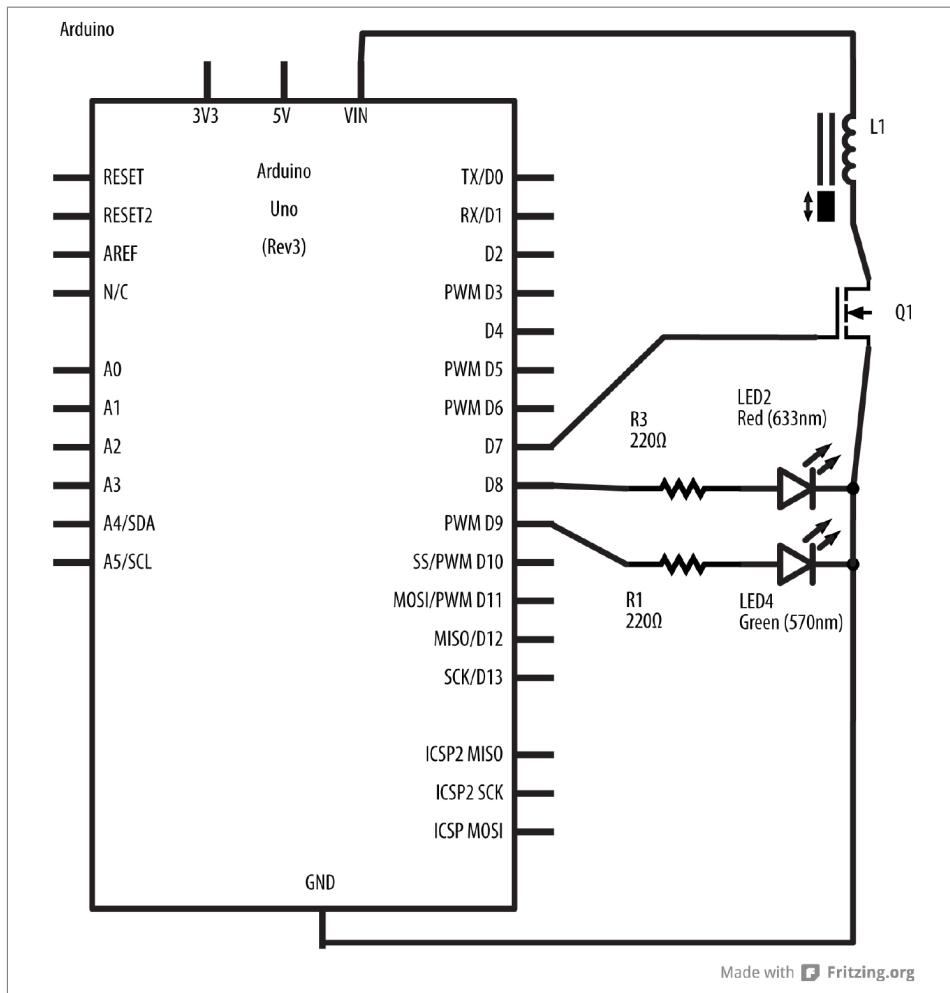


图7-10：NFC读取器设备的电路示意图

跟你做的写入器类似，开始Arduino读取器sketch（剪影），需要包含库、常量和全局变量。除了实例化NfcAdapter，你还需要创建一个Time库的实例。使用常量来为输出引脚和锁的房间号赋值，然后从卡的信息中添加数据到变量，比如入住时间、结账时间，以及最后一张卡的读取是不是好的。

```
// If you're using an SPI-based shield, change these to include the SPI library:  

#include <Wire.h>  

#include <PN532_I2C.h>  

#include <PN532.h>  

#include <NfcAdapter.h>  

#include <Time.h>
```

```
PN532_I2C pn532_i2c(Wire);
```

```
NfcAdapter nfc = NfcAdapter(pn532_i2c);

const int lockPin = 7;          // pin that the solenoid door lock is on
const int greenLed = 9;         // pin for the green LED
const int redLed = 8;           // pin for the red LED
const long roomNumber = 3327;   // the room number

time_t checkin = 0;            // checkin time
time_t checkout = 0;           // checkout time
String cardName = "";          // name on the card
long cardRoomNumber = 0;        // room number on the card
long readTime = 0;              // last time you tried to read a card
boolean goodRead = false;
```

setup () 函数仍然初始化输出引脚，并初始化nfcAdapter和串行通信。同时也设置时间，请务必更改时间值，这样才能反映出当前时间：

```
void setup() {
    Serial.begin(9600);
    // set the clock to the date & time
    // hour (24-hour format), minute, second, day, month, year.
    // e.g. date shown here is Jan 9 2013, midnight:
    setTime(00, 00, 00, 1, 9, 2013);

    nfc.begin();           // initialize the NFC reader
    pinMode(lockPin, OUTPUT); // make the door lock pin an output
    digitalWrite(lockPin, LOW); // set it low to lock the door
    pinMode(greenLed, OUTPUT); // make pin 9 an output
    pinMode(redLed, OUTPUT); // make pin 10 an output

    Serial.println(F("\nHotel NDEF Reader"));①
    Serial.print(F("Current Hotel Time is "));
    Serial.println(formatTime(now()));
    Serial.print(F("This is the lock for room "));
    Serial.println(roomNumber);
}
```

①Serial.print (F ())；表示将常量字符串写到闪存中以保留程序内存。

这个循环则检查自从最后一次读卡后一共过去了多少毫秒，并使用millis () 函数返回自从上一次Arduino被重置后过去了多少毫秒。

如果小于3s（3000ms），则锁定并指示LED开或者关，具体是开还是关，则取决于最后一次读卡的状态。

如果大于3s（自从最后一次读卡之后，不管是成功的还是失败的），这个循环就会调用一个函数来监听标签：

```
void loop() {
    if (millis() - readTime < 3000) { // less than three seconds since last tag
        digitalWrite(greenLed, goodRead); // green LED lights for a good read
        digitalWrite(lockPin, goodRead); // lock opens if you get a good read
        digitalWrite(redLed, !goodRead); // red LED lights if you don't
    }
    else { // after three seconds, lock again
        digitalWrite(greenLed, LOW); // turn off green LED
        digitalWrite(redLed, LOW); // turn off red LED
        digitalWrite(lockPin, LOW); // lock door
        goodRead = listenForTag(); // listen for tags
    }
}
```

`listenForTag()` 函数仅仅是使用 `NfcAdapter` 的 `tagPresent()` 函数去监听标签，如果标签被扫描到，并且有 NDEF 消息，则用 `for` 循环读取里面的记录。在这个例子中，消息里只有一条记录。一旦你知道了记录的长度，就可以设置一个字节数组去保存它，并使用 `getPayload()` 函数将记录拷贝到数据中。一旦你将字节数组转换为 `String` 类型，通过解析它就能得到相关的字段，这样你才能知道客户的姓名、房间号、入住时间和结账时间。后面会提到一个 `parsePayload()` 函数，它能很方便地帮你解析出这些数据。

```
boolean listenForTag() {
    boolean unlockDoor = false;
    resetValues();

    if (nfc.tagPresent()) { // if there's a tag present
        readTime = millis(); // timestamp the last time you saw a card
        NfcTag tag = nfc.read();
        if (tag.hasNdefMessage()) { // every tag won't have a message
            NdefMessage message = tag.getNdefMessage();
            NdefRecord record = message.getRecord(0);

            if (record.getTnf() == TNF_MIME_MEDIA &&
                record.getType() == "text/hotelkey") {
```

```

// Get the length of the payload:
int payloadLength = record.getPayloadLength();
byte payload[payloadLength]; // make a byte array to hold the payload
record.getPayload(payload);

// convert the payload to a String
String json = "";
for (int c=0; c< payloadLength; c++) {
    json += (char)payload[c];
}
parsePayload(json); // parse the payload

// check if you can let them in or not:
unlockDoor = isValidKey();
}

}

return unlockDoor;
}

```

要解析有效荷载的相关数据，你需要遍历记录，并从JSON对象中找到所包含的键/值对。先找到开和关括号的位置（JSON数据由大括号包裹），在这个数据里面，键/值对之间用逗号分隔，键/值对中间则用冒号。所以，如果你知道当前逗号、最后一个逗号以及中间冒号的位置，你就能提取出每个键/值对了。最后一个键/值对没有逗号，所以当你找不到逗号的时候，这个键/值对就是最后一个了。

```

void parsePayload(String data) {
    // you only care about what's between the brackets, so:
    int openingBracket = data.indexOf('{');
    int closingBracket = data.indexOf('}');
    // your individual data is between two commas:
    int lastComma = openingBracket;
    int comma = 0;
    // parse the data until the last comma:
    while (comma != -1){
        String key, value;
        int colon = data.indexOf(':', lastComma); // get the next colon
        comma = data.indexOf(',', colon); // get the next comma
        // key is between the last comma and the colon:
        key = data.substring(lastComma+1, colon);

        // if there are no more commas:
        if (comma == -1) { // value is between colon and closing:
            value = data.substring(colon+1, closingBracket);
        }
    }
}

```

```
    }
} else {           // value is between colon and next comma:
    value = data.substring(colon+1, comma);
}
```

一旦你得到了键/值对，就需要去掉引号。String.replace () 可以将引号替换为空字符串，然后再使用String.trim () 来去掉空字符串。

```
// now to get rid of the quotation marks:
key.replace("\"", " ");   // replace any " around the key with spaces
key.trim();               // trim away the spaces
value.replace("\"", " "); // replace any " around the value with spaces
value.trim();             // trim away the spaces

// now, look for the possible data you care about:
setValue(key, value);
lastComma = comma;
}
}
```

一旦你得到了键/值对，就创建一个名为setValue () 的函数来检查键值并赋值给对应的全局变量。姓名的值需要被保存为String类型，房间号则是Integer类型，入住时间和结账时间都是time_t类型（就是32位的整数），这两个时间将用来比对当前时间。String类有一个很方便的函数将字符串转换为整数：

```
void setValue(String thisKey, String thisValue) {
    if (thisKey == "checkout"){
        checkout = thisValue.toInt();
    }
    else if (thisKey == "checkin") {
        checkin = thisValue.toInt();
    }
    else if (thisKey == "name") {
        cardName = thisValue;
    }
    else if (thisKey == "room") {
        cardRoomNumber = thisValue.toInt();
    }
}
```

在这个sketch（剪影）中你需要的另外一个函数是isValidKey（），它在listenForTags（）函数中被调用。它主要将入住时间和结账时间与当前时间进行对比，如果当前时间处于这个客户在酒店住的时间范围内就返回true，否则返回false。

```
boolean isValidKey() {  
    boolean result = false;  
    if (cardRoomNumber == roomNumber) {  
        if (now() <= checkin) {  
            Serial.println("You haven't checked in yet.");  
            Serial.println("Current time " + formatTime(now()));  
            Serial.println("Your arrival " + formatTime(checkin));  
        }  
        else if ((now() >= checkin) && (now() <= checkout)) {  
            Serial.println("Welcome back to your room, " + cardName + ".");  
            result = true;  
        }  
        else if (now() >= checkout) {  
            Serial.println("Thanks for staying with us! You've checked out.");  
            Serial.println("Current time " + formatTime(now()));  
            Serial.println("Your departure " + formatTime(checkout));  
        }  
    }  
    else {  
        Serial.print("This card can't unlock room ");  
        Serial.print(roomNumber);  
        Serial.println(".");  
    }  
    return result;  
}
```

在这个函数中有大量的串行消息，但是它们仅仅用于调试目的，因为实际的设备没有串行接口。如果你想节省内存，在知道这个sketch（剪影）可以正常工作的情况下，可以注释掉这些串行消息。唯一能够对用户界面造成影响的只有result被设置为true还是false。以下是一个没有串行消息的简易版本：

```
boolean isValidKey() {  
    boolean result = false;  
  
    if (cardRoomNumber == roomNumber) {  
        if ((now() >= checkin) && (now() <= checkout)) {  
            result = true;  
        }  
    }  
    return result;  
}
```

```
    }
}
return result;
}
```

在这个sketch（剪影）中还有一个有效的函数，它将时间变量格式化为一个字符串进行打印。如果你想尽量减少代码，那么这个函数和调用都可以注释掉，前提是你知道程序能正常工作。

```
String formatTime(time_t time) {
    TimeElements elements;
    breakTime(time, elements);
    String formatted = "";
    formatted += elements.Month;
    formatted += "/";
    formatted += elements.Day;
    formatted += "/";
    formatted += elements.Year + 1970;
    formatted += " ";
    formatted += elements.Hour;
    formatted += ":";
    formatted += elements.Minute;
    return formatted;
}
```

完整的源代码可以在GitHub (<http://bit.ly/hoteltagreader>) 上找到。

你可以在设置功能时重新设置时间，并且只要所设定的时间处于你写入标签的时间之间，锁就会打开。设置三次不同的参数并上传到这个sketch（剪影），一次选择之前的时间，一次中间的时间，再一次选择之后的时间，看看会发生什么。如果你使用的是例子最后显示的实例JSON，那么你的入住时间是2013年1月7日，结账时间为2013年7月10日。

Arduino NDEF写入器设备的浏览器接口

正如你所看到的，将客户信息作为JSON对象输入到串行端口监视器不是件容易的事情，所以你需要编写一个桌面应用程序，这样就能通过串行端口将信息发送到标签写入器设备。如果能写一个简单的HTML页面，通过页面输入的形式将信息发送到串行端口那样更好。

大多数桌面程序语言都提供了一个外部库来访问计算机的硬件端口。然而，Web浏览器一般是不提供该功能的，这样就能保护电脑不受外部网站代码的损害。但是，当这样的网页服务器应用程序驻留在你的计算机上时，你就能比较安全地访问硬件端口了。

Node.js有很多与PhoneGap相似的概念，两者都能让你在操作系统上运行JavaScript程序，而不仅仅是运行在浏览器上。两者都允许你用JavaScript和HTML写应用程序，并且还能利用本地库提供的功能。Node.js是一个运行在OS X、Windows和Linux上本地应用程序的JavaScript解释器，任何本地应用程序能做的，它都可以做到，包括访问串行端口、A/V硬件等。为了从丰富的HTML输入中获益，你可以使用Node.js编写运行在电脑上的JavaScript服务器程序，它能在浏览器中提供用户界面，并且也能从串行端口读取并发送数据。

如果你已经通过了前面的章节，那么你已经安装好了Node.js，并使用它的包管理器（npm）安装好了Cordova-CLI工具。现在是时候使用它为你的标签读取器设备编写用户接口了。

使用Node.js你需要两个外部库：一个是称为express.js的Web框架，一个是名为nodeserialport的串行端口库。express.js能让你通过RESTful结构构建Web服务器应用程序。每个URI元素都能被用来定义程序功能。serialport能让你从计算机的串行端口读取数据和写入数据。

Node.js的应用规范： package.json

Node和npm使用一个名为package.json的文件来描述给定节点的应用程序的依赖。该文件描述了应用程序、它的依赖，以及它依赖的引擎。这个文件位于应用程序的目录下，在你启动它之前包管理器就使用它在子目录下为应用程序安装所有的库。应该感谢它，这样你只需要发布程序源代码和Web用户接口文件（HTML、JavaScript、CSS和图片等）。对于你要构建的应用程序，package.json看起来像这样：

```
{  
  "name": "HotelNodeApp",  
  "version": "0.1.0",  
  "description": "A browser-to-serial application",  
  "keywords": "serial, node-serialport",  
  "author": {  
    "name": "Tom Igoe"  
  },  
  "dependencies": {  
    "serialport": "1.1.x",  
    "express": "3.x"  
  },  
  "engines": {  
    "node": "0.10.x",  
    "npm": "1.2.x"  
  }  
}
```

在这个例子中，包文件最重要的部分是列出了你需要的Node.js的最低版本和npm，以及库依赖及其版本。你还需要一个node使用格式的版本号。

使用此应用程序的目录，并将这个保存为package.json。你需要的其他文件是：index.js（主文件）和index.html（用来创建用户接口），其余部分将通过npm安装。

客户端代码

这个应用程序的index.html文件相当简单。有一个需要你填写字段的表单：姓名、房间号、你住的天数。入住时间是文档头的script自动生成的，并填充到这个表单的隐藏字段里。图7-11所示为浏览器视图。下面是HTML文件。

```
<!DOCTYPE html>
<html>
  <head>
    <script type="text/JavaScript">

      function tick() {
        var now = Date(),
            clock = document.getElementById("clock"),
            checkin = document.getElementById("checkin");
        clock.innerHTML = now.toString();
        checkin.value = now;
      }

    </script>
  </head>
  <body onload="setInterval(tick, 1000);">
    <form action="/submit" method="post">
      Name: <input type="text" name="name"><br />
      Room: <input type="text" name="room"><br />
      Checkin time: <span id="clock"></span><br>
      <input type="hidden" name="checkin" id="checkin" >
      Number of days: <input type="text" name="days"><br />
      <input type="submit" value="submit">
    </form>
  </body>
</html>
```

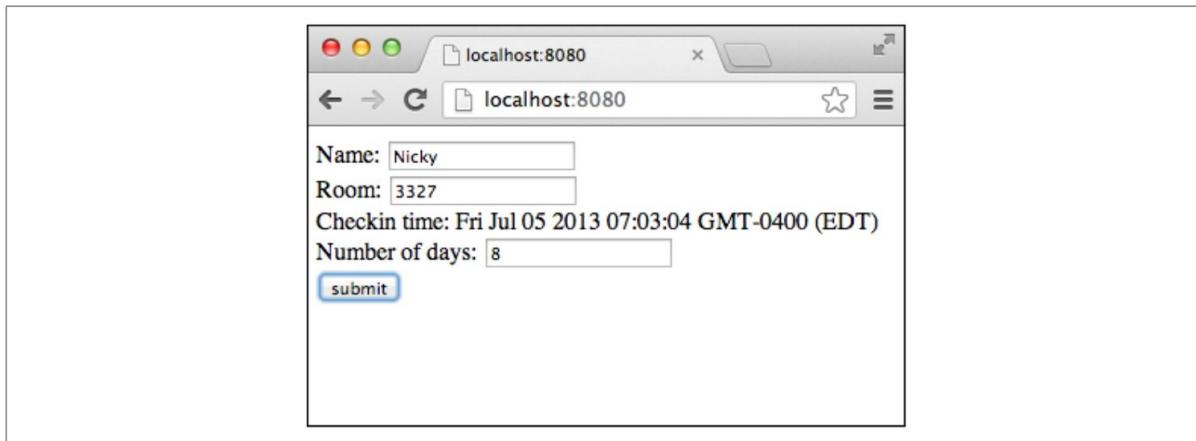


图7-11：办理酒店入住手续的用户界面浏览器视图

服务器端代码

在服务器端的JavaScript（index.js）中，首先声明了一些变量。它实例化了serialport库和express库，然后使用express创建了一个HTTP服务器来处理请求，并使用从命令行调用的第三方令牌打开串行端口（通过输入node index.js portname命令行调用，其中portname是到串行端口的路径——后面会有更详细的介绍）。最后两个变量用来保存通过串行端口从Arduino传入的酒店记录：

```
var serialport = require("serialport"),    // include the serialport library
  SerialPort = serialport.SerialPort,      // make a local instance of serial
  express = require('express'),           // make an instance of express
  app = express(),                      // start Express framework
  server = require('http').createServer(app), // start an HTTP server
  portName = process.argv[2],            // third token of the command line
  record = {},                          // NDEF record to send
  deviceMessage = "";                  // messages from the writer device
```

接下来，需要告诉express.js你要使用它的解析器中间件。这可以让你很容易地解析从页面得到的HTTP请求主体，之后启动服务器（注意，它运行在8080端口，而不是平常的HTTP服务器端口80）并打开串行端口，在命令行上让用户知道：

```
// use the bodyParser middleware for express:
app.use(express.bodyParser());
server.listen(8080);      // listen for incoming requests on the server
console.log("Listening for new clients on port 8080");

var myPort = new SerialPort(portName, {    // open the serial port
  // look for newline at the end of each data packet:
  parser: serialport.parsers.readline("\n")
});
// print the port you're listening on:
console.log("opening serial port: " + portName);
```

这个串行端口的实例调用解析器允许你告诉串行端口什么时候产生一个事件。这是最简单的从串行端口读取输入的串行数据，如果这

个端口为每个传入的文本行生成一个事件，那么解析器就会查找换行符“\n”。

该程序的其余部分包括三个事件处理程序：一个用于处理输入的串行数据，两个用于处理传入的HTTP请求。串行数据处理程序查找输入的串行数据的新行，如果该行包含Result:，那么它会将其复制到全局变量之一deviceMessage：

```
// listen for new serial data:  
myPort.on('data', function (data) {  
    // for debugging, you should see this in the terminal window:  
    if (data.search("Result:") != -1) {  
        deviceMessage = data;  
    }  
    console.log("Received: " + data);  
});
```

接下来的两个处理程序来自express.js并用于处理HTTP GET和POST请求。app.get () 返回index.html页面的任何GET请求，所以http://localhost.com: 8080、http://localhost.com: 8080/index.html、http://localhost.com: 8080/foo或者你在浏览器上的任何输入都会返回这个索引页面：

```
// respond to web GET requests with the index.html page:  
app.get('*', function (request, response) {  
    response.sendFile(__dirname + '/index.html');  
});
```

index页面中的表单通过最后的处理程序app.post () 处理POST请求。这个处理程序由两部分组成，在第一部分，从表单元素中提取数据并放入记录变量中，时间则在本部分结束时完成转换处理，因为JavaScript的日期对象是以毫秒为单位的，Arduino的Time库会很快地处理它，跟你在前面的Arduino例子中看到的一样。当所有必要的记录部分都从表单中提取出来并计算完成后，处理程序将会把这些数据发送到串行端口。

```
// take anything that begins with /submit:  
app.post('/submit', function (request, response) {  
    record.name = request.body.name; // get the name from the body
```

```
record.room = request.body.room; // get the room number from the body
var days = request.body.days; // get the number of days from the body
var today = new Date(request.body.checkin); // get the time from the body
// calculate the checkout timeStamp:
var departure = new Date(today.valueOf() + (days * 86400000));
// convert to unix time in seconds:
record.checkin = Math.round(today.valueOf()/1000);
record.checkout = Math.round(departure.valueOf()/1000);
// send it out the serial port:
myPort.write(JSON.stringify(record) + "\n");
```

连接到串行端口的Arduino NFC写入器设备需要时间来读取数据，然后将其写入到标签后才做出回应。所以app.post () 第二部分的处理是账户的破解位 (bit of hack)，它是一个HTTP响应的开始，包括HTML页面，这样足以告诉客户发生了什么并给她一个连接返回到表单。然后，它使用setTimeout () 打开连接保持3秒，等待串行端口的响应，包括NFC写入器设备的响应。如果在规定的时间内没有响应，则关闭端口。

```
// write the HTML head back to the browser:
response.writeHead(200, {'Content-Type': 'text/html'});
// send the data:
response.write("<p><a href=\"^\">Return to form</a></p>");
response.write("Sent the following to the writer device:<br>");
response.write(JSON.stringify(record) + "<p>");

// wait 3 seconds before closing the connection, so that
// you can get a response from the writer:
setTimeout(function() {
    // if you got a response from the writer, send it too:
    if (deviceMessage != "") {
        response.write("response from writer device: " + deviceMessage + "<p>");
        deviceMessage = "";
    } else {
        response.write("no tag present");
    }
    // send the link back to the index and close the link:
    response.end();
}, 3000); // end of setTimeout()
}); // end of app.post()
```

如果你正在开发一个完整的UI，则可以有一些客户端JavaScript代码来接收JSON字符串并以可读的方式呈现出来。但是对于现在来说，

你应该按照NFC的原则来编写代码。

完整的源代码可以在GitHub (<http://bit.ly/hotel-node-app>) 上找到。

这是应用程序的所有代码，一旦保存了index.js文件，你就需要导入库。要做到这一点，你应该在命令行中切换到程序的目录（如果此时不在程序的目录下）。在这个目录下应该有三个文件：

- package.json
- index.html
- index.js

要导入库，则输入：

```
$ npm install
```

你应该看到一些响应行，从一系列npm的GET请求相关文件开始，最后结尾看起来则像两个编译的总结：

```
express@3.3.3 node_modules/express
├── methods@0.0.1
├── fresh@0.1.0
├── range-parser@0.0.4
├── cookie-signature@1.0.1
├── buffer-crc32@0.2.1
├── cookie@0.1.0
├── debug@0.7.2
├── mkdirp@0.3.5
├── send@0.1.2 (mime@1.2.9)
├── commander@1.2.0 (keypress@0.1.0)
└── connect@2.8.3 (uid2@0.0.2, pause@0.0.1, qs@0.6.5, bytes@0.2.0, ...)

serialport@1.1.1 node_modules/serialport
├── bindings@1.1.0
├── sf@0.1.6
├── async@0.1.18
└── optimist@0.3.7 (wordwrap@0.0.2)
```

如果你看到了“NOT OK”，不要理它们，这意味着出了些问题，如果通过npm响应回滚，你就会发现问题。最常见的问题就是没有使用最新版本的node、npm或库（你可以通过更新package.json来修复它）；没有网络连接，或者不兼容库和操作系统。对于后者，你需要检查问题列表或者在论坛上找到问题的解决方案。

Node-serialport和OS X



在OS X 10.8.x（山狮）及以后版本中，在安装node-serialport上可能会有一些问题。你需要安装XCode命令行工具。Apple并没有让这些便利的工具很容易找到，但是你可以登录Apple开发者网站(<https://developer.apple.com/xcode/>)进行下载，你只需要正常登录就能做这件事，并不需要支付开发者授权费用。

一旦你安装了这些库，就会在应用程序的目录下创建一个名为node_modules的新的子目录，这是库文件所在的位置，现在你可以测试应用程序了。

首先确保你已经使用Arduino IDE将标签写入器sketch（剪影）上传到Arduino。注意Arduino串行端口的名字，然后在命令行输入：

```
$ node index.js portname
```

对于portname，使用你的串行端口的名称，就跟你在“Arduino开发环境”部分学到的一样。你应该得到这样的回应：

```
Listening for new clients on port 8080
opening serial port: /dev/tty.usbmodem621
```

现在打开浏览器访问http://localhost: 8080，你应该得到如图7-11所示的页面。将标签写入器装上，并输入你的信息，则应该得到一个返回页面，类似于图7-12。



图7-12：办理酒店入住手续的用户界面响应页面的浏览器视图

总结

现在，你已经完成了所有的用户界面，以及完成了一家酒店从入住到门锁的登记系统的构建。恭喜你！下面回顾一下所有的组件：

在Arduino Uno和NFC shield间有两个物理设备（说不定你已经在自己的例子中重复使用了Uno和shield），第一个物理设备是NFC写入器，它接收JSON字符串并将其写入标签。这个设备连接到笔记本电脑或者台式电脑，电脑中运行你刚刚编写的节点应用程序。节点应用程序在浏览器中提供了入住用户界面，有客户的名字、房间号、自动生成的时间和日期，以及客户将要住的天数，然后生成一个JSON字符串给写入器。

第二个物理设备是门锁，当客户将标签移到设备上时，它将检查标签上的客户房间号、入住时间，结账时间和当前时间并以此来判断门是否开启，如果数据合法，那么就打开电磁锁。

这使NFC在物理环境下的广泛应用成为可能。正如你所看到的，在微型控制器上使用NFC跟在手机和桌面电脑环境下使用一样。即使微型控制器的编程环境比操作系统所提供的功能更有限，但是它们仍然提供相同的基本功能：读写标签，解释NDEF消息和记录。它们还可以感测和控制物理世界，这是计算机和移动设备不能够直接执行的能力。

为了在NFC应用中更广泛地使用计算设备，你必须考虑它的长处和局限性，然后选择合适的工具。例如，微型控制器在这个门锁应用中就是一个非常不错的选择，你同样可以将这个登记程序放到平板装置上，或者是后面你将看到的基于USB的NFC读取器上。当你需要建立一个自定义物理接口时，你应该尝试设计并编程实现最快的响应速度和给定条件下的清晰度。对于每个物理事件，像限制阻断函数的使用和提供LED自动调节等小细节，将使得设备的感知可靠性有很大的不同。

在下一章中，你将超越简单的读取和写入标签功能，真正区别RFID和NFC：设备之间的对等通信。

第8章 点对点（P2P）交换

到目前为止，你在本书中见到的所有例子都是用被动式标签模式来进行NDEF数据交换的。这是对RFID标准的一个很大的改进，因为它引入了涵盖一系列不同标签类型（虽然不是所有的RFID标签类型）的通用数据格式。RFID和NFC之间的另一个主要区别是后者的P2P交换能力。这种交换是直接在两个有源设备之间进行的。之前学习的内容，NFC和RFID都不是在对等网络环境中使用的。在本章中，你将看到如何编写P2P交换的例子。

每次P2P交换，都有一个目标方和一个发起方，两者都是有源设备。这意味着当你通过P2P发送NDEF消息给目标方时，目标方也可以在处理这些数据后给你发送反馈信息。你还可以从一个标签中复制信息，将其存储在设备上，并将其传输到另一个设备或其他标签上。P2P提供了一系列有趣的功能。

P2P交换使用另一个NFC协议：简单的NDEF交换协议（SNEP）。SNEP是一个请求-应答协议：发起方发送一个想交换某种数据的请求，目标方对此请求做出应答。SNEP基于NFC论坛的逻辑链路控制协议（LLCP）。虽然在编写应用程序时你不必直接与SNEP和LLCP打交道，但理解了它们在NFC架构中的位置和作用后对你的开发工作也是有帮助的，它也有助于解释P2P和标签读写的差异。正如从图8-1你所看到的，SNEP和LLCP层用于数据打包和P2P传输。这层是设备控制器和NDEF层之间的关键环节。

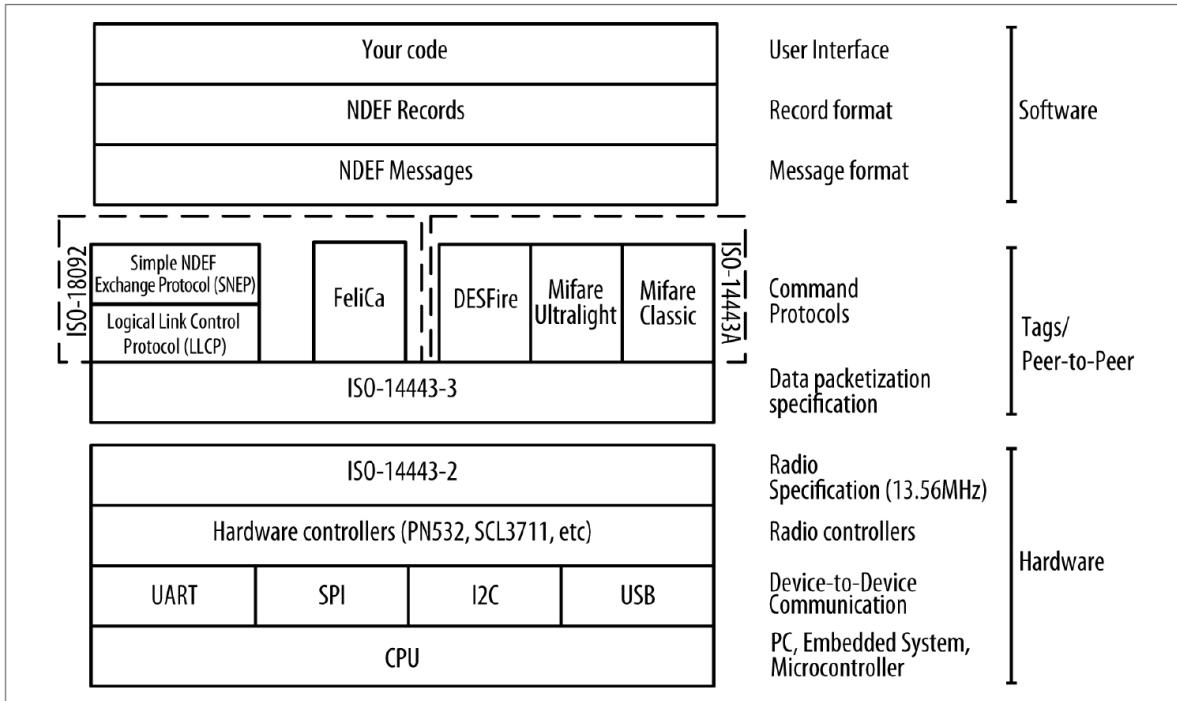


图8-1：SNEP和LLCP作为数据链路层适配进NFC架构，执行类似于标签格式化的功能

SNEP和LLCP没有定义该由操作系统或用户如何处理P2P交换。每个操作系统都以自己的方式实现了SNEP和LLCP。Android的当前实现被称为Android Beam（之前的Android 4.0，Android有一个实现称作NDEF推送协议，或NPP）。当你想用Beam在两个Android设备之间交换数据时，让它们背靠背贴着，随后会出现“Touch to beam”界面。当你在发送方点击发送时，它就发起与接收设备间的P2P数据交换，交换的数据被封装到一个或多个NDEF消息里发送。Beam被设计成让你在图8-1所示的堆栈上层进行操作。它处理P2P逻辑链路的数据，并把NDEF消息交给分发系统。

当你启用NFC时，Beam激活NFC射频并寻找来自其他设备或标签的连接。在第5章中你已经知道了当一个标签接近而没有前台应用程序来处理它时，标签分发系统就处理它。Beam处理P2P也类似。如果你的设备上的NFC探测到另一个P2P设备，Android就会弹出“Touch to beam”界面。图8-2显示了这个界面。如果你的应用程序实现了P2P分享功能（Android Java框架的NDEF推送协议），然后选择任何你想分享的数据，这些数据就被发送到目标设备上了。如果没有实现分享功能，那么Android将发送应用程序的URI来启动应用程序。如果目标设

备上有相应的应用程序，系统就会打开它。如果没有，Android将打开Google Play应用程序商店进去查找。

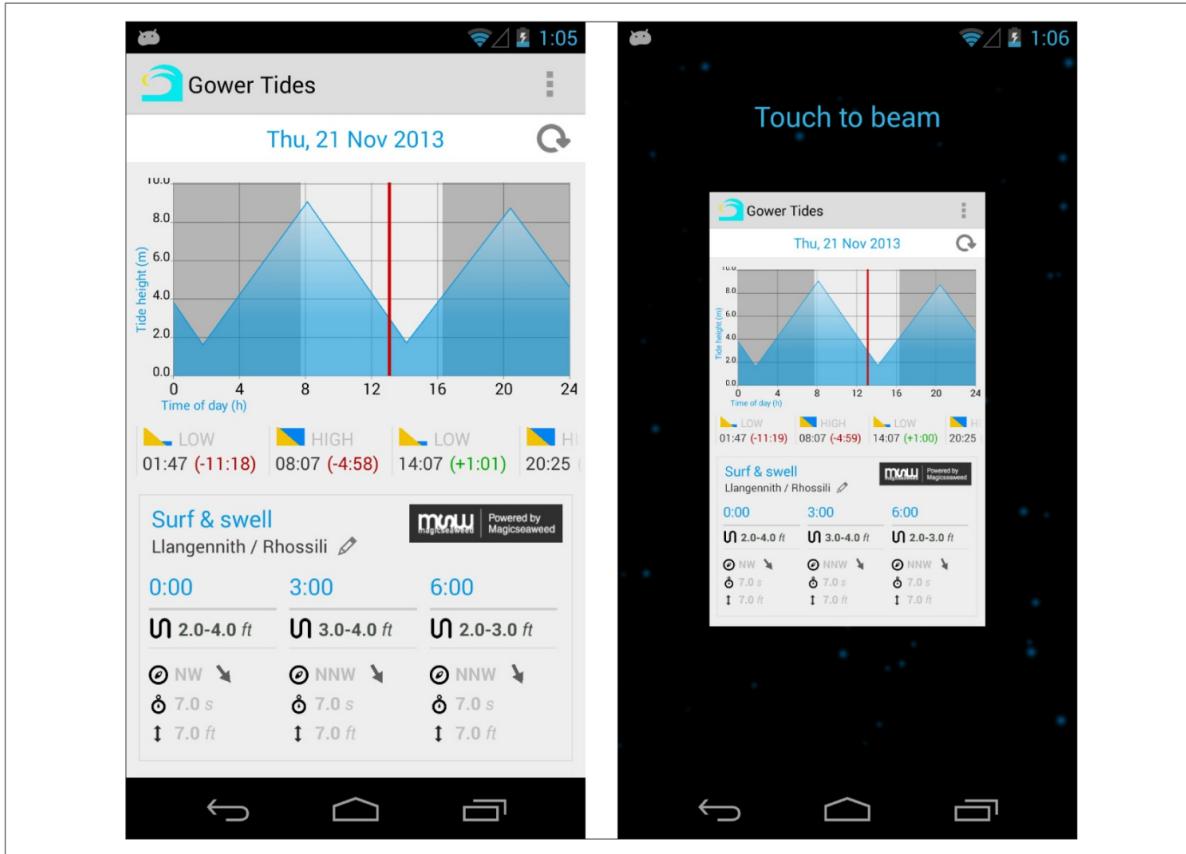


图8-2：“Touch to beam”界面（左图是Gower Tides应用程序，右图是应用程序在用Beam）

这是Android Beam不同于Android上以前P2P实现的地方，也是不同于其他系统上P2P实现的地方。在某些情况下，默认自动弹出“Touch to beam”界面是比较恼人的，要关闭这个默认设置，需要在AndroidManifest.xml文件的<application>标记后添加如下行：

```
<application android:debuggable="true" android:hardwareAccelerated="true"
    android:icon="@drawable/icon" android:label="@string/app_name">
    <meta-data android:name="android.nfc.disable_beam_default"
        android:value="true" />
    <activity android:configChanges=
        "orientation|keyboardHidden|keyboard|screenSize|
        locale" android:label="@string/app_name" android:name="P2P"
        android:theme="@android:style/Theme.Black.NoTitleBar">
```

在PhoneGap中发送P2P消息

理解P2P的最好办法就是尝试一下，为此你需要准备两个具有NFC功能的Android设备。如果只是看默认的行为，那么你甚至都不需要编写一个新的应用程序。你可以使用NXP TagInfo，或者用在第5章中所写的NDEF读取器应用程序。在目标设备上打开一个应用程序，在发起设备上打开另一个应用程序。把两个设备背对背贴着，然后轻触发起设备的屏幕。在目标设备上，你应该能得到发起设备上前台应用程序的URI信息。图8-3显示了发起设备上的Gower Tides通过beam到目标设备上的NXP TagInfo的结果。当一个未实现P2P的应用程序被beam时，Android Beam将传送一个包含两条记录的信息给它：一条是TNF 01记录，内含Play Store上该应用程序的URI信息；另一条是TNF 04记录，内含应用程序的Android应用记录（AAR）。

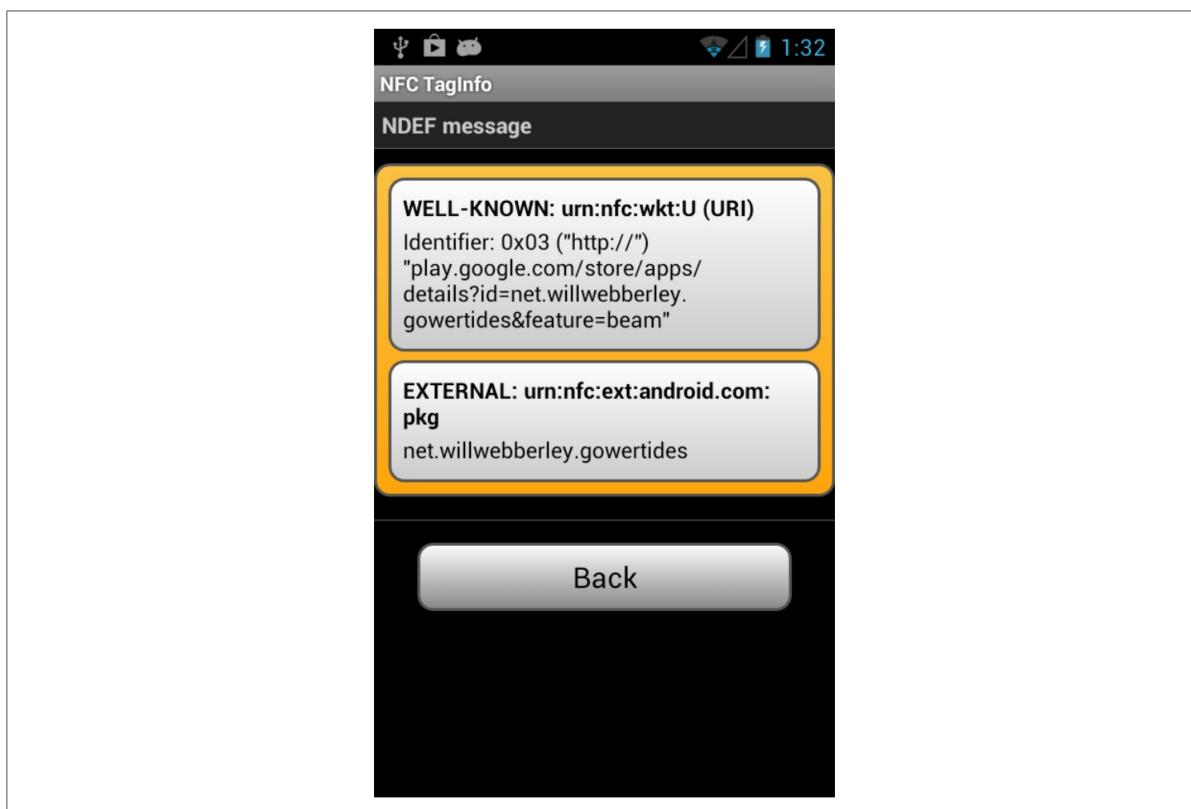


图8-3：用Android Beam从Gower Tides发送默认消息；应用程序没有实现P2P，但Android Beam给它发送了一个NDEF消息

当你想重写默认行为，并发送自己的Android NDEF消息时，则可以设置一个NDEF推送消息或设置一个Beam推送URI。在Android系统里这两种方式都有相应的回调选项。在PhoneGap-NFC插件中，NDEF推送只要用nfc.share()，而URI Beam只需nfc.handover()。当你分享一个消息时，它就推送，目标方成功接收该消息后，则会提示你成功接收到。

下一个应用程序使用了一些之前用到的NDEF消息，在前面的章节中这些消息被写入标签，这次通过P2P接收消息而不是通过读取标签，比较一下有啥不同。在大多数情况下，你会看到没什么区别。你也可以使用这个程序通过P2P发送自定义消息，看看它们的响应。

你需要用到之前使用过的工具：

- 两个具有NFC功能的Android设备
- Android SDK
- 一个文本编辑器
- 在计算机上安装Cordova CLI
- Node.js和npm也要安装好

开始创建一个新的项目：

```
$ cordova create ~/P2P com.example.p2p P2P ①
$ cd ~/P2P②
$ cordova platform add android
$ cordova plugin add https://github.com/don/phonegap-nfc
```

①Windows用户应该用%userprofile%\P2代替~/P2P。

②Windows用户应该用/d%userprofile%\P2代替~/P2P。

当项目已经准备好后，打开www目录中的index.html和index.js文件。

index.html页面有一个简单的输入新的NDEF记录的表单和一个下拉菜单，这个下拉菜单让你从几条预设的记录里选择一条记录。后面，你会在index.js文件中填充这个菜单。

```
<!DOCTYPE html>
<html>
  <head>
    <title>PhoneGap NFC P2P</title>
  </head>
  <body>
    <h2>PhoneGap NFC P2P</h2>
    <div class="app">
      <div id="messageDiv"></div>
      <form>
        <select id="sampleField"></select><br />
        <input type="hidden" id="kindField" /><br/>
        <div id="typeDiv">
          Type:<br/>
          <input type="text" id="typeField" value="" size="30" />
        </div>
        Payload:<br/>
        <textarea id="payloadField" rows="10" cols="30"></textarea>
        <br/>
      </form>
    </div>
    <script type="text/JavaScript" src="cordova.js"></script>
    <script type="text/JavaScript" src="js/index.js"></script>
    <script type="text/JavaScript">
      app.initialize();
    </script>
  </body>
</html>
```

index.js文件开始处有一个名为data的数组用以保存几条JSON编码的NDEF记录。这些记录中的大多数你已在之前的例子中见过。它们将作为各种不同类型的记录发送出去。

```
var data = [
  {
    name: "Text Record",
    kind: "text",
    data: "hello, world"
  },
  {
```

```

        name: "URI Record",
        kind: "uri",
        data: "http://oreilly.com"
    },
    {
        name: "Address",
        kind: "mime",
        type: 'text/x-vCard',
        data: 'BEGIN:VCARD\n' +
            'VERSION:2.1\n' +
            'N:Coleman;Don;;\n' +
            'FN:Don Coleman\n' +
            'ORG:Chariot Solutions;\n' +
            'URL:http://chariotsolutions.com\n' +
            'TEL;WORK:215-555-1212\n' +
            'EMAIL;WORK:don@example.com\n' +
            'END:VCARD'
    },
    {
        name: "Hue Settings",
        kind: "mime",
        type: 'text/hue',
        data: JSON.stringify({
            "1": {
                "state": {
                    "on": true, "bri": 65, "hue": 44591, "sat": 254
                }
            },
            "2": {
                "state": {
                    "on": true, "bri": 254, "hue": 13122, "sat": 211
                }
            },
            "3": {
                "state": {
                    "on": true, "bri": 255, "hue": 14922, "sat": 144
                }
            }
        })
    },
    {
        name: "Android Application Record",
        kind: "external",
        type: "android.com:pkg",
        data: "com.joelapenna.foursquared"
    },
    {
        name: "Empty",
        kind: "empty",
        data: ""
    }

```

```
    }  
};
```

下面是主程序部分。开始的几个函数看起来很眼熟：一个 initialize () 、一个bindEvents () 和一个onDeviceReady () 。

```
var app = {  
  /*  
   Application constructor  
  */  
  initialize: function() {  
    this.bindEvents();  
    console.log("Starting P2P app");  
  },  
  /*  
   bind any events that are required on startup to listeners:  
  */  
  bindEvents: function() {  
    document.addEventListener('deviceready', this.onDeviceReady, false);  
    sampleField.addEventListener('change', app.showSampleData, false);  
    // modify the form so it doesn't generate a submit event:  
    document.forms[0].onsubmit = function(evt) {  
      evt.preventDefault(); // don't submit  
      payloadField.focus(); // put the payload field in focus  
    };  
    // if either type or payload is changed, update the share:  
    typeField.onchange = app.shareMessage;  
    payloadField.onchange = app.shareMessage;  
  },  
  
  /*  
   this runs when the device is ready for user interaction:  
  */  
  onDeviceReady: function() {  
    var option;  
  
    // populate the sampleField from the data array  
    sampleField.innerHTML = "";  
    for (var i = 0; i < data.length; i++) {  
      option = document.createElement("option"); // make an option element  
      option.value = i; // give it this number  
      option.innerHTML = data[i].name; // get the data object  
      if (i === 0) { // select the first element  
        option.selected = true;  
      }  
      sampleField.appendChild(option); // add this to sampleField
```

```
    }

    app.showSampleData();
},
```

接下来是P2P函数。shareMessage () 函数查找当前表单里的数据记录，格式化为NDEF消息，并使用nfc.share () 通过P2P分享出去。

```
/*
  Share the message from the form via peer-to-peer:
*/
shareMessage: function () {
  // get the MIME type, and payload from the form
  // and create a new record:
  var payloadType = typeField.value,
      payloadData = payloadField.value,
      kind = kindField.value,
      record;
  app.clear();           // clear the message div
  app.display("Publishing message"); // display the notification
  // use a different ndef helper to format the message
  // depending on the kind:
  switch (kind) {
    case "text":
      record = ndef.textRecord(payloadData);
      break;
    case "uri":
      record = ndef.uriRecord(payloadData);
      break;
    case "mime":
      record = ndef.mimeMediaRecord(payloadType, payloadData);
      break;
    case "external":
      record = ndef.record(ndef.TNF_EXTERNAL_TYPE, payloadType, [], ...);
      break;
    case "empty":
      record = ndef.emptyRecord();
      break;
    default:
      alert("ERROR: can't build record");
  }

  console.log(JSON.stringify(record));
  // share the message:
  nfc.share(
    [record],           // NDEF message to share
```

```
function () {      // success callback
    navigator.notification.vibrate(100);
    app.display("Success! Message sent to peer.");
},
function (reason) { // failure callback
    app.display("Failed to share message " + reason);
});
},

```

当一个消息被目标方成功接收后，发起方会被回调一个成功的通知，你将得到消息“成功！消息已发送给对端”。你可以使用此回调来触发其他行为。

nfc.share () 函数有两个可选的回调，对应于success和failure。当目标设备确认收到消息后success函数被回调。这是一个很有用的通知方式，告诉发起方目标方已准备好接收新消息。

unshareMessage () 函数用来关闭分享功能。在此应用程序中，并没有使用到这个函数，但它仍然显示在这里，你可以看看它的样子：

```
/*
Stop sharing:
*/
unshareMessage: function () {
// stop sharing this message:
nfc.unshare(
    function () {          // success callback
        navigator.notification.vibrate(100);
        app.clear();
        app.display("message is no longer shared");
},
    function (reason) { // failure callback
        app.display("Failed to unshare message " + reason);
});
},

```

只有一个UI相关的函数。当下拉菜单变化时，它会调用 showSampleData () 来获取相应的日期元素，并回填到表单字段：

```
/*
Get data from the data array and put it in the form fields:

```

```
/*
showSampleData: function() {
    // get the type and payload from the form
    var index = sampleField.value,
        record = data[index];

    // fill form with the data from the record:
    kindField.value = record.kind;
    typeField.value = record.type;
    payloadField.value = record.data;

    // hide type for kinds that don't need it
    if (typeof record.type === 'string') {
        typeDiv.style.display = "";
    } else {
        typeDiv.style.display = "none";
    }

    app.shareMessage();
},

```

最后是熟悉的display () 和clear () 函数，用于写入message div：

```
/*
    appends @message to the message div:
*/
display: function(message) {
    var label = document.createTextNode(message),
        lineBreak = document.createElement("br");
    messageDiv.appendChild(lineBreak);      // add a line break
    messageDiv.appendChild(label);          // add the text
},
/*
    clears the message div:
*/
clear: function() {
    messageDiv.innerHTML = "";
},
}; // end of app

```

你只需要在两个设备中的一个上运行此应用程序。当运行它时，会得到如图8-4所示的界面。你可以在下拉菜单中选择任何一条记录作为数据分享出去，或者自己写条新的记录。要分享信息，可以选择任

何预设记录，或者自己写，然后将运行此应用程序的设备和第二个设备背对背贴着。如果第二个设备上装有Foursquare和Mood Setter应用程序，那么当你分享这些信息到相应的应用程序时，它们将会被打开。该地址记录将触发Android系统，问你是否将记录保存到联系人表中。

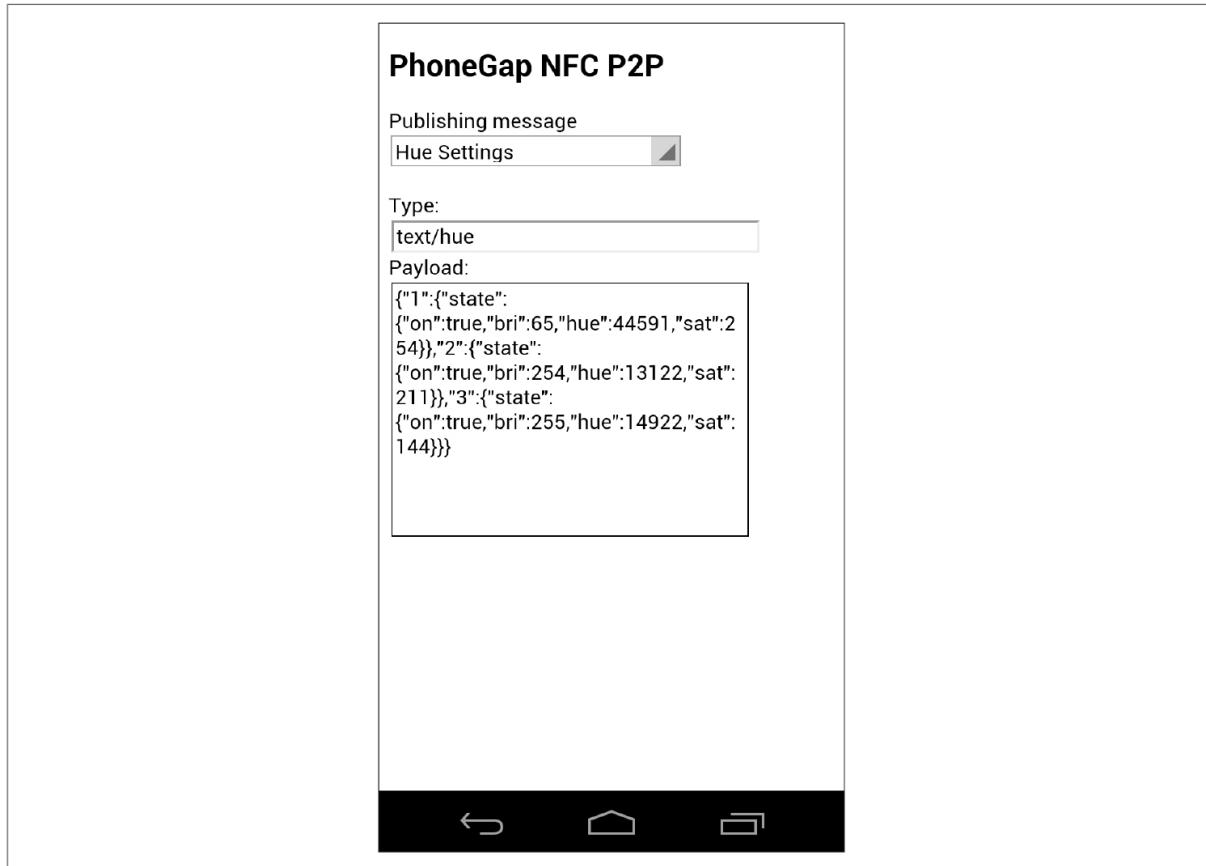


图8-4：你的第一个P2P应用程序

在PhoneGap中接收P2P消息

接收P2P消息和接收基于标签的消息类似。你只需要添加一个或多个NFC事件监听器。当前台应用程序收到NDEF推送消息时，就像处理基于标签的消息那样处理它。

要让应用程序增加接收能力，你需要再添加一个事件监听器。你至少会收到两种不同类型（MIME类型或external类型）的NDEF消息，所以如果想用一个监听器捕获这两种类型，则需要一个NdefListner。在onDeviceReady（）函数中添加以下代码：

```
nfc.addNdefListener(  
    app.onNfc,           // nfcEvent received  
    function (status) {   // listener successfully initialized  
        app.displayMessage("Listening for NDEF messages.");  
    },  
    function (error) {    // listener fails to initialize  
        app.displayMessage("NFC reader failed to initialize "  
            + JSON.stringify(error));  
    }  
);
```

然后添加一个名为onNfc（）的函数来处理此事件，你可以把它放到程序的任何地方。下面的代码是节选的，如果你想看完整代码，则可以看第5章的例子。

```
/*  
 * displays info from @nfcEvent in message div:  
 */  
onNfc: function(nfcEvent) {  
    // if there is an NDEF message on the tag, display it:  
    var thisTag = nfcEvent.tag,  
        thisMessage = thisTag.ndefMessage,  
        tagData = "";  
    // display the tag properties:  
    tagData = "Tag ID: " + nfc.bytesToHexString(thisTag.id) + "<br />"  
        + "Tag Type: " + thisTag.type + "<br />"
```

```
+ "Max Size: " + thisTag.maxSize + " bytes<br />"  
+ "Is Writable: " + thisTag.isWritable + "<br />"  
+ "Can Make Read Only: " + thisTag.canMakeReadOnly + "<br />";  
if (thisMessage != null) {  
    // get and display the NDEF record count:  
    tagData += "<p>Tag has NDEF message with " + thisMessage.length  
    + " records.</p>";  
}  
app.displayMessage(tagData);  
},
```

保存应用程序并运行。当你分享消息从一个设备到另一个设备时，新的代码将监听这些消息，并显示这些信息。

完整的源代码可以在GitHub (<http://bit.ly/master-p2p>) 上找到。

你可能已经注意到了，和第5章的例子相比，我们只保留了标签类型和标签UID的显示。这个交换中没有使用到物理标签，所以会得到一系列空结果：

```
Tag ID: 00  
Tag Type: android.ndef.unknown  
Max Size: 0 bytes  
Is Writable: false  
Can Make Read Only: false
```

如前所述，标签被数据链路层的LLCP和SNEP层所替代，所以标签ID、类型、大小和其他metadata在P2P交换中都是无意义的。但这不要紧，应用程序仍然能得到NDEF消息。你可以重组这个程序来删除标签metadata功能。

P2P消息和标签消息的处理方式相同，这非常有用，因为这意味着你可以在应用程序中设计相同的交互界面。

切换

P2P分享类似于短消息，但是当你想传输大文件，如音频文件或照片时，它就不是很适合了。当交换文件时，必须保持两个设备贴在一起。所以如果要传输一个大文件，则可能要把这两个设备贴在一起很长一段时间。这是不方便的。

NFC连接切换规范被设计用来处理这种情况。当接收到切换消息时，NFC库检查操作系统，看看是否有更好的传输介质，如蓝牙或WiFi，并尝试用新方式传数据。

切换的工作原理是：发起方发送一条以切换请求记录（TNF: Well-Known类型，RTD: Handover Request ("Hr")）开头的NDEF消息，后跟一些替代载体记录（TNF: Well-Known类型，RTD: Alternative Carrier ("ac")）。替代载体记录是发起方设备上可供选择的不同传输媒介，如蓝牙、WiFi或其他。记录的载荷就是替代载体的MAC地址。

当目标方收到这个消息后，回复一个切换选择记录（TNF: Well-Known类型，RTD: ("Hs")）和一套自己的替代载体记录。通常列出的载体顺序反映了目标方的偏好。然后，发起方选择一个替代的载体，并反馈一个切换载体消息（TNF: Well-Known类型，RTD: ("Hc")）确定替代载体。目标方然后答复切换载体消息，内含配置数据和/或载体凭证，从而可以在没有蓝牙配对、WiFi登录或其他认证情况下直接进行数据传输。

正如你所看见的，建立切换的协商过程涉及几次消息交互。为了保证切换成功，在切换协商过程中要保持设备始终接触着，直到协商过程完成并切换到新载体，否则将失败。

切换过程是不可思议的。没有配对工作要做，没有密码要进行交换。即使设备的蓝牙或WiFi没打开也不要紧，NFC协议栈会向操作系

统请求将其打开并进行交换。甚至切换协议还有省电机制，让目标方和发起方协商替代载体的功率以延长电池寿命。

但是切换也有一些潜在的问题。如果你选择的替代载体正在被其他程序使用时，那么切换可能会失败。例如，你正通过蓝牙放着音乐，并尝试通过蓝牙发送一个文件时，那么传输速度明显变慢，并且传输有时会失败。如果你的设备离开了载体的有效范围，传输也将失败。

正如前面提到的，每个平台都以自己的方式实现P2P。例如，Beam能在Android与BB10之间工作，但有一个额外的步骤，以接受远程设备的配对请求。SNEP是潜在的跨平台协议，应该是任何操作系统都能实现以简化配对，但依然有问题。

静态切换

NFC论坛提供了一种切换方法来管理NFC设备与非NFC设备之间的数据交换。如果非NFC设备附有一个被动NFC标签，标签内含有一条切换选择记录，用以指示替代载体的相关配置数据，那么NFC设备将尝试切换到替代载体与非NFC设备进行通信。例如，假设你想在一个Android NFC设备与iPhone之间进行数据交换，如果iPhone附有一个NFC标签，标签内有iPhone上蓝牙或WiFi配置数据的切换选择消息，那么Android设备可以通过NFC获取配置数据后，尝试切换到蓝牙或WiFi进行连接后交换数据。

这不是一个完美的解决方案，因为它依赖于非NFC设备上的替代载体开着并准备好，否则连接将失败，或者iPhone用户必须显式启用该连接。那时还不如一开始就直接在两个设备之间启用蓝牙或WiFi来交换数据，免去麻烦的静态切换。理想的用例可能是iPhone或iPad驱动kiosk与用户的NFC设备进行通信。

在PhoneGap中发送切换消息

在PhoneGap-NFC插件中，Android的Bean URI方法，用于管理切换，被简化为nfc.handover ()。它的API类似于nfc.share ()，但它分享一个文件，而不是一个NDEF消息。下面的示例演示了如何使用它。你可以选择发送文件，或者拍摄照片并发送。你需要之前使用过的工具：

- 两个具有NFC功能的Android设备
- Android SDK
- 一个文本编辑器
- 在计算机上安装Cordova CLI
- Node.js和npm要安装好

开始创建一个新的项目：

```
$ cordova create ~/FileSender com.example.filesender FileSender①
$ cd ~/FileSender ②
$ cordova platform add android
$ cordova plugin add https://github.com/don/phonegap-nfc
$ cordova plugin add https://github.com/don/cordova-filechooser
```

①Windows用户应该用%userprofile%\FileSende代替~/FileSender。

②Windows用户应该用/d%userprofile%\FileSende代替~/FileSender。

在这里，你要使用的一个额外的插件是filechooser。之所以不用PhoneGap的文件API，是因为PhoneGap的文件API不太好用，在之前的例子中你也应该体会到了。filechooser插件简化了文件选择操作，它利用Android自带的文件选择器来选择文件，并返回文件的URI。

index.html文件里除了通常的message div外，还增加了几个元素：一个拍照按钮、一个选择文件按钮和一个用于显示拍摄照片的div：

```
<!DOCTYPE html>
<html>
  <head>
    <title>File Sender</title>
  </head>
  <body>
    <h1 id="cameraButton">Take Picture</h1>
    <h1 id="filePicker">Choose File</h1>
    <div id="messageDiv"></div>
    <div id="photoDiv"></div>
    <script type="text/JavaScript" src="cordova.js"></script>
    <script type="text/JavaScript" src="js/index.js"></script>
    <script type="text/JavaScript">
      app.initialize();
    </script>
  </body>
</html>
```

index.js文件以通常的方式开始，将事件处理函数绑定到UI元素：

```
var app = {
  /*
  Application constructor
  */
  initialize: function() {
    this.bindEvents();
    console.log("Starting File Sender app");
  },

  /*
  binds events that are required on startup to listeners.
  */
  bindEvents: function() {
    // bind events to the UI elements:
    document.addEventListener('deviceready', this.onDeviceReady, false);
  },

  /*
  runs when the device is ready for user interaction.
  */
  onDeviceReady: function() {
    cameraButton.addEventListener('touchstart', app.takePicture, false);
    filePicker.addEventListener('touchstart', app.chooseFile, false);
  },
};
```

filePicker和拍照按钮的按键处理程序会调用Android系统相应的系统应用程序，操作完成后，返回所选择文件的URI，或者刚才拍摄的照片文件。takePicture（）函数还能设置相机参数并拍照。你可以从PhoneGap文档页面（<http://docs.phonegap.com/>）中的“Camera”栏目了解更多有关拍照API的信息：

```
/*
brings up the file chooser UI:
*/
chooseFile: function() {
    fileChooser.open(
        app.onFileSystemSuccess, // success handler
        app.failure           // failure handler
    );
},
/*
Brings up the camera app:
*/
takePicture: function () {
    navigator.camera.getPicture(
        app.onCameraSuccess,   // camera capture success handler
        app.failure,          // failure handler
        {
            // image capture options
            quality: 75,
            destinationType: Camera.DestinationType.FILE_URL,
            sourceType: Camera.PictureSourceType.CAMERA,
            targetWidth: 300,
            targetHeight: 300,
            correctOrientation: true,
            saveToPhotoAlbum: false
        }
    );
},
```

一旦成功，则调用其相应的success处理程序。onFileSuccess（）处理函数只调用shareMessage（）。onCameraSuccess（）处理函数在调用shareMessage（）前在页面上显示照相机图像。它们使用相同的failure（）处理函数：

```
/*
When you get a good picture, share it:
*/
onCameraSuccess: function (imageURI) {
```

```

var img = document.createElement("img");
img.src = imageURI;          // add the URI as the img src
photoDiv.innerHTML = "";      // clear old image
photoDiv.appendChild(img);    // add the image element to the photoDiv
app.display(imageURI);       // show the URI
app.shareMessage(imageURI);  // share the image
},
/*
When you get a good file, share it:
*/
onFileSystemSuccess: function (fileURI) {
  photoDiv.innerHTML = "";
  app.display(fileURI);
  app.shareMessage(fileURI);
},
/*
When you fail to get a file or photo, cry:
*/
failure: function (evt) {
  console.log(evt.target.error.code);
},

```

下面是你一直等待的部分：shareMessage（）。这个函数和上个例子中那个相同名字的函数功能几乎一样。这个函数用文件的URI作为参数，它也进行检查，看URI里是否包含空格（有`%20`时退出URI）。这是因为Android Beam API有个bug阻止发送有空格的URI，即使是escaped时。在这个应用程序的一个全功能版本中，遇到有空格的URI时会先做文件的本地备份，重新命名文件后发送该副本。但现在，你将不得不选择没有空格的发送文件。

```

/*
Share the URI from the file or photo via P2P:
*/
shareMessage: function (uri) {
  // Android Beam API has a bug that prevents sending files
  // with spaces in the URI:
  if (uri.search("%20") > 0) {
    app.clear();
    app.display("Sorry. Can't beam a URI with spaces. Android Beam Bug");
    return;
  }

  app.clear();
  app.display("Ready to beam " + uri);
}

```

```
app.display("Place your device back to back with another device to beam.")
// beam the file:
nfc.handover(
  uri,
  function () {           // success callback
    navigator.notification.vibrate(100);
    // you know when the beam is sent and the other device received
    // the request but you don't know if the beam completes or fails
    app.display("Success! Beam sent.");
    app.unshareMessage(); // unshare the file when complete
  },
  function (reason) {      // failure callback
    app.clear();
    app.display("Failed to share file " + reason);
  }
);
},
```

既然有一个shareMessage () 处理函数，那么必须有一个unshareMessage () 处理函数。这和前面的应用程序一样。接下来的是display () 、 clear () 和clearAll () 以清除照片div。

```
/*
Turns off sharing
*/
unshareMessage: function () {
  // stop beaming:
  nfc.stopHandover(
    function () {           // success callback
      navigator.notification.vibrate(100);
      app.display("File is no longer shared");
      setTimeout(app.clearAll, 5000); // clear the screen after 5 seconds
    },
    function (reason) {      // failure callback
      app.display("Failed to unshare file " + reason);
    }
  );
},
/*
appends @message to the message div:
*/
display: function(message) {
  var label = document.createTextNode(message),
  lineBreak = document.createElement("br");
  messageDiv.appendChild(lineBreak); // add a line break
  messageDiv.appendChild(label); // add the text
}
```

```
},
/*
clears the message div:
*/
clear: function() {
    messageDiv.innerHTML = "";
},
clearAll: function() {
    app.clear();
    photoDiv.innerHTML = "";
}
}; // end of app
```

完整的源代码可以在GitHub (<http://bit.ly/master-flesender>) 上找到。

当运行这个应用程序时，你会发现在Beam运行之前就得到了一个成功的消息。该消息来自初始化切换请求和确认（见图8-5）。如果你能保持看到这个消息，则通常说明文件传送成功。如果使用切换发送文件有麻烦，则请确保蓝牙连接（或其他替代载体）此时没有做其他的事情。

切换并不只是为了传输大文件，它也能作为一种方便的设备配对方式来使用。例如，HomeSpot的NFC蓝牙耳机，通过轻触设备就能完成蓝牙配对，然后音频流就通过蓝牙发送到耳机了。这种配对方式可以用在许多不同的消费电子设备上，以此把手机或平板电脑变成遥控器。除了配对外，还可以在初始化切换请求时包含一个控制器应用程序的URI给对方，让对方下载。

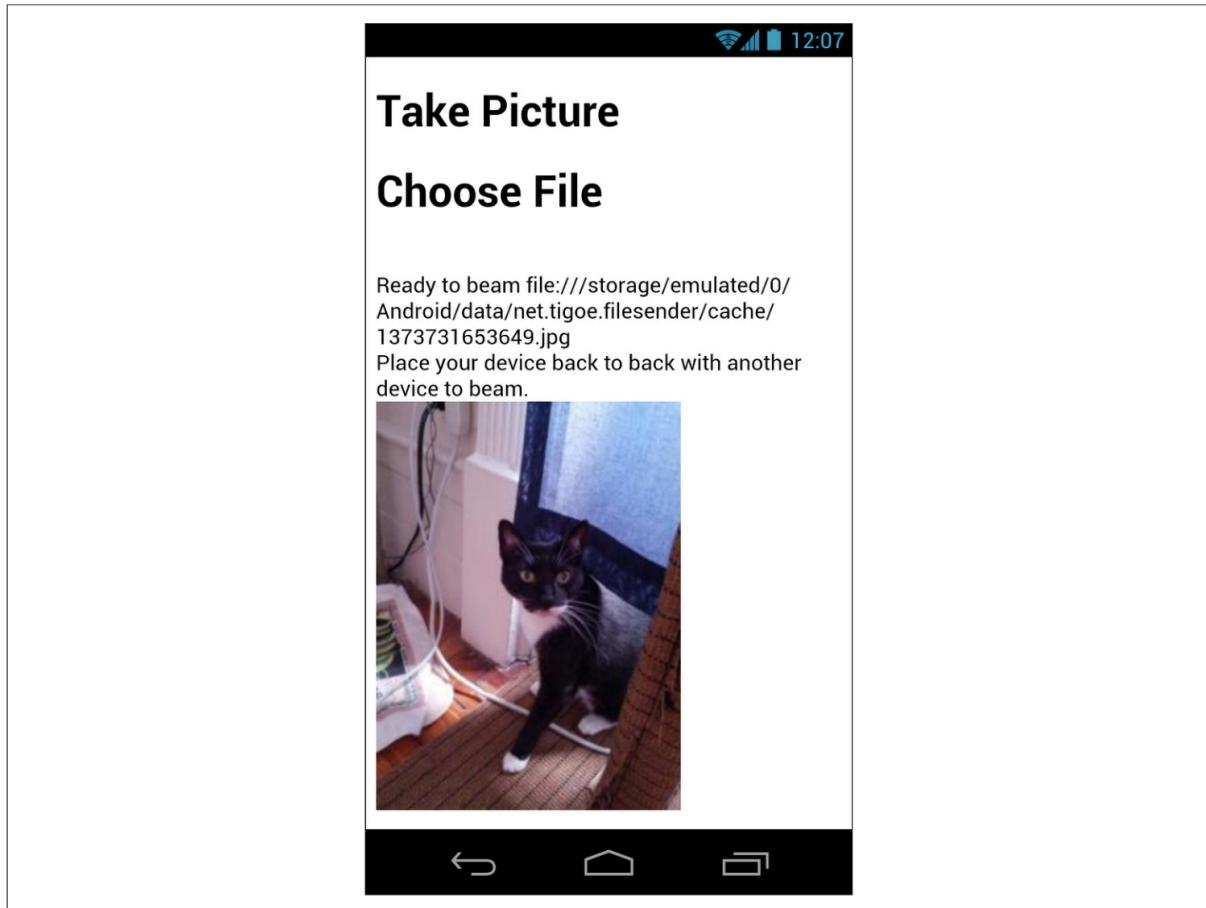


图8-5：FileSender应用程序演示切换

用Arduino进行P2P

在Arduino上使用第7章提到的NFC shield是可以P2P的，但有些复杂。在撰写本文时，还没有库做到抽象SNEP和LLCP层后，能让你只关注于NDEF交换。Arduino上的P2P库已经由Michael Weir完成，并且可以在GitHub (<http://bit.ly/embed-pn532>) 上找到。虽然这个库缺少Arduino风格的抽象API，也需要开发者更深入的C知识，然而，理论上它可以作为一个简单库的基础。

Seeed Studio已经做了一些事情来整合NDEF库与Weir的工作。他们的成果也可以在GitHub (<http://bit.ly/nfc-dev>) 上找到。但还有更多的事情要做，以完全抽象SNEP和LLCP层。

到目前为止，P2P仅工作在SPI上，所以要修改Adafruit shield以适配这些库。你需要将跳线焊接在shield的SEL0和SEL1上。你还需要从MISO、MOSI、SS和SCK跳线到所选择的SPI初始化引脚上，以适配Weir或SeeedStudio库。

现在，我们使用的Arduino库都不支持切换。切换需要一个额外的蓝牙shield、模块或一个WiFi shield作为替代载体，并有一个集成了NFC和Bluetooth或WiFi的库。这些很容易让Arduino Uno的可用程序内存溢出，但在Arduino Due上是有可能实现的。

卡仿真

除了读写标签和P2P通信模式外，NFC设备还有标签仿真模式。在此模式下，NFC设备响应其他NFC设备时，就像它是一个NFC标签。这意味着此设备就像一个非接触式智能卡。

这个功能有一些有用的应用场景。当把设备当智能卡用时，读取器设备能验证卡（或卡仿真）上的数据，不仅确保数据是真实的，而且与数据相关的卡ID也是准确的。例如，如果你的卡中有银行账号，并且仿真卡和银行账户卡具有相同的UID，自动取款机或收款机就可以做可靠的交易了。标签仿真就是被设计用于这样的金融交易或售票的。这就是谷歌钱包和其他系统底层的技术，让你的手机像一张现金卡或信用卡。

为了使标签仿真安全，采用这种技术的银行和其他客户需要确信这个系统是安全的。所以NFC论坛规定NFC读写器硬件必须包含安全元件或安全接入模块。这只是在NFC控制器里附加了一个简单的处理器。安全元件可以被编程使用公开密钥系统加密和解密算法来验证由读取器发送的代码的正确性。因为读写器（或智能卡）的安全元件可以运行它自己的小程序（叫applet或taglet），它甚至可以动态生成新的哈希值。如果你有完全控制硬件的权限，则当然可以按照自己所希望的任何模式来配置安全模块。

设计卡仿真功能的部分原因是，通过让NFC设备模拟各种智能卡（信用卡、员工ID/门禁卡和公交卡等），你可不再需要随身携带所有这些卡了。虽然理论上可行，但在实践中有很多困难。这意味着一些利益相关者有机会参加进来，如手机制造商、移动服务提供商、手机软件开发商、各大银行等。银行、安全系统供应商、公共交通运营商等都需要接入手机上的NFC安全模块，以模仿他们的卡。虽然技术上可行，具体操作起来却很复杂。

接入安全元件通常由硬件供应商限制权限，需要与软件或固件供应商签订协议后才能接入。现在Android标准版本里没有接入NFC设备上安全元件的API。如果你是美国运通，决定实现基于手机的交易，那么谷歌和三星将竭诚与你合作，有可能很快就获得接入安全元件的许可协议。但如果你只是一个位于伊利诺伊州皮奥里亚的两人的工艺品店，则可能会等上一段时间。

因为Android是开源的，你可以找到定制版的Android系统，它开放接入NFC安全元件的接口。这种方案的接入依赖于操作系统和设备上特定的硬件模块，所以现在没有一个通用的解决方案。

Adafruit和Seeed Studio shield上的NXP PN532模块有一个安全元件，它是可配置的。但在写本文时，Arduino库里还没有PN532的安全接入模块（SAM）的编程接口函数。本书中提到的库只是简单地把SAM设置成正常工作模式，作为一个读写器用，并不执行卡仿真。

总结

NFC的P2P交换增加了一个很有价值的组成部分——直接从一个设备发送NDEF消息到另一个设备的能力，这是简单的RFID不能提供的。因为P2P交换的复杂性，它主要限于在智能设备上使用，如Android、BlackBerry或Windows Phone设备。但愿，Arduino上的P2P和其他简单的微型控制器上的P2P在不久的将来将有所提高。同时，本书的最后一章将让你尝试一下用NFC库在嵌入式Linux设备上做个例子。

第9章 嵌入式系统里的NFC

到目前为止，我们已经看到了移动应用程序和物理交互程序里的NFC例子。除了Android设备平台外，也有其他嵌入式设备平台的NFC解决方案，同样，除了Arduino外也有其他物理交互平台的解决方案。最有趣的是嵌入式Linux系统，如由博通（Broadcom）提供支持的德州仪器（TI）的BeagleBone和Raspberry Pi平台项目。这两个平台的开发板售价不到50美元，嵌入式Linux系统运行在ARM处理器上。它们都提供直接访问CPU的物理输入输出引脚的能力，这个功能使得跨这两种板子开发项目更方便。在本章中，你将了解到NFC技术在这些平台上的最新进展，并简要地介绍用Node.js为这些平台编写NFC应用程序。

正如你在Arduino章节看到的，嵌入式Linux系统上的NFC工具现状远没有像PhoneGap-NFC插件那样发展充分，甚至也不如用于Android、BlackBerry或Windows手机上的标准NFC API。可用的库通常比较底层，可以提供接入到NFC堆栈底部层次的信息。但有很多从事嵌入式开发的技术人员对NFC很感兴趣，所以这个领域的相关工具发展很迅速。



这一章会有很多命令行的内容。你会看到很多CLI的安装、配置和构建的指令。如果你以前没有接触过基于CLI的编译器和包管理器，则可能会发现一些不熟悉的术语。在之前的章节中你已经接触过Cordova CLI，但在本章中你会看到更多的指令。虽然我们已经尽了最大努力来解释这些内容，但如果你是一个Linux编程工具的新手，则可能还是需要一些其他的参考材料来辅助学习。

嵌入式Linux设备和包管理器介绍

德州仪器的Beagle板、BeagleBone、BeagleBone Black和BeagleBoard MX都是基于TI的ARM AM335x、AM35x和AM37x处理器的嵌入式Linux板，它们是专门为嵌入式爱好者设计的。它们皮实、价格低廉，且在网上 (<http://beagleboard.org/>) 有很多公开可用的硬件设计例子。TI还推出了一个Ångström Linux嵌入式开发的定制版本，但可以运行Ubuntu、Android和其他嵌入式Linux发行版。对于下面的例子，我们建议用BeagleBone Black。

Raspberry Pi板分A型和B型，都是基于Broadcom公司的ARM处理器。不同于BeagleBone，Pi的设计不向公众开放，但是安装和购买信息可以在<http://www.raspberrypi.org/>上找到。Pi模型运行一个称为Raspbian的Debian Linux发行版的定制版本。在本章的例子中，我们使用Adafruit的Raspbian版本称为，Occidentalis。像BeagleBoard板，Pi也可以运行其他嵌入式Linux系统。A型板是B型板的简装版：一个USB接口，没有以太网接口，只有一半的内存。为了达到最佳效果，你需要一个B型板。

除了前面列出的网站，你也可以在Adafruit的教程网站 (<http://learn.adafruit.com>) 找到使用Raspberry Pi或BeagleBone Black的教材。Matt Richardson和Shawn Wallace写的Getting Started with Raspberry Pi，以及Matt Richardson写的Getting Started with BeagleBone，都是不错的入门书。

网络、USB和NFC

在开始工作前有三个重要的准备工作需要完成：

- Terminal已接入到板子上（你可以通过串行terminal session或SSH terminal session来实现）
- 板子能连上互联网（你需要下载包，用于SSH）
- 一个USB接口用于NFC适配器

不同的Pi和BeagleBone型号的硬件特性会有所不同，你在挑选时需要考虑下面几点：

- Raspberry Pi和BeagleBone都没内置WiFi。如果你无法通过有线以太网连接板子，那么就需要一个USB到WiFi的适配器。
- Raspberry Pi的A型板、BeagleBone和BeagleBone Black只有一个USB接口。这意味着当你把USB做NFC适配器用时，将没有WiFi USB适配器可用。
- Raspberry Pi的A型板没有有线以太网接口，所以你只能通过WiFi 连接到Internet。A型板是所提到的唯一一个不能同时上网并使用NFC适配器的板子。
- Raspberry Pi（两种型号）没有内置的USB转串口的接口。它需要一个3.3V的USB转串口的适配器，用于连接到计算机的串口。BeagleBone型号都有USB转串口的内置USB mini-B连接器。
- Raspberry Pi默认通过键盘和屏幕进行设置。如果按照默认方式进行设置，你需要准备一个显示器、HDMI连接线和键盘。但是，你也可以用一个USB转串口适配器来进行设置。你还可以通过命令行方式来设置，只是工作量会多一点。详细内容请参见“Raspberry Pi特性”一节。
- Raspberry Pi一般很容易设置NFC软件，但设置硬件会复杂点。在测试中，我们发现Raspberry Pi和BeagleBone Black在易用性、性能和价格方面大致相当，特别是当考虑到必要的扩展性，如电缆和电力供应时。

虽然市场上各种嵌入式Linux开发板提供不同的硬件功能和处理器，以及使用不同的Linux版本，但有一些通用功能它们都提供。它们都能联网，当然，它们都有一个命令行界面。它们都支持像USB和UART串口标准的硬件接口。最重要的是，只要有必要的存储空间、内存和硬件组件，它们都可以运行为Linux编写的软件。它们中的大多数也支持少量的编程语言。至少，它们都支持C++，这就是很多你用过的包都有一个C++内核的原因，即使并不需要使用它。为了方便代码开发，所有主流的操作系统，甚至你在这里使用的基本Linux版本，也支持包管理系统。在本章中，你不需要自己写C++代码，但是要安装和使用几种不同的工具编译一些代码。

软件开发人员利用库和工具来高效完成他们的工作，并且不同的软件项目有不同的依赖关系。在这里你会看到，一个基于USB的NFC读取器项目将依靠其他软件库来支持USB通信。优秀的开发人员会列出项目的所有依赖关系，当然，利用包管理系统这个工具使得该管理工作变得更容易。在之前的章节中，你已经用过一个包管理工具——node包管理器（npm）。Cordova不是真正的包管理器，它更多的是自动化项目构建工具，但是它正朝着带有Cordova CLI的插件工具的包管理器方向发展。APT、opkg、npm、苹果的App Store和谷歌的Play Store，都是包管理器。App市场包管理器和开发者工具包管理器的区别在于，后者一般要在电脑上安装编译器和编译源代码，然后安装到设备上；而前者是将预编译的二进制文件直接下载和安装到设备上。

Linux、UNIX和其他POSIX操作系统上的包管理系统使得安装和编译软件库的操作过程独立于操作系统本身。文件结构和API在所有操作系统上都是非常相似的，所以Linux上的Raspberry Pi代码（或多或少）就可以在Mac OS X上使用。因此，在本章中我们提供的例子是通用的Linux版本的代码，并假定可以让它们在嵌入式Linux板上工作。这些例子我们已经在Raspberry Pi和BeagleBone Black上测试过，必要时，我们会提示这两个平台的特异性。



本章中你不会看到适用于Mac OS X或Windows的任何例子。这里的重点是在嵌入式世界。

对于本章接下来的内容，你需要准备：

- BeagleBone、Raspberry Pi或其他嵌入式Linux板
- 1A或以上的电源
- SLC3711非接触式USB智能卡读取器 (<http://bit.ly/1fBYLhm>)
(我们的NFC适配器)
- 一些Mifare Classic的NFC标签 (在本章中，Mifare Classic是和所使用的Linux库最兼容的，尽管它们在一些Android设备上没法工作；请参见第2章中的“设备与标签类型匹配”一节了解更多的关于哪些设备与哪些标签能更好地协作)
- 一个文本编辑器

可选但有用的是：

- 一个适合电路板的USB WiFi适配器 (Adafruit的Miniature WiFi (802.11b/g/n) Module (<http://www.adafruit.com/products/814>) 效果很好)
- 一个支持NFC适配器的A到A USB扩展器
- 一个USB到TTL串行电缆——调试/控制台电缆 (<http://www.adafruit.com/products/954>) (确保是工作在3.3V，而不是5V)



如果板子是从电脑的USB接口供电，那么这里提到的NFC读取器和可选的WiFi适配器会消耗过多主板上的电流。当你尝试读取标签时，由于供电不足，将会得到很多误读，所以你应该至少提供1A的电源，最好是2A的独立电源。

因为每个板子的设置各有不同，有些比较复杂，所以我们将跳过板子设置部分。你应该遵循板子的操作指导来设置并连接到Internet，并熟悉登录到命令行。本章的其余部分将假设你可以登录到板子上的命令行，也可以通过板子连接到Internet。

这里提到的USB智能卡读取器是一个有USB接口的NFC读取器。它的天线较弱，所以你会发现它阅读标签时会有些麻烦，除非定位准确。我们发现USB延长线是个很好的助手，可以让读取器更靠近标签。

嵌入式Linux上的NFC：概览

既然现在市场上没有内置NFC的嵌入式Linux板，那么你就需要一个单独的NFC控制器。你可以使用PN532 shield，但它们没有设计与BeagleBone或Pi的接口，所以基于USB的模块更易于使用。在本章中，我们选择了Identive SLC3711非接触式USB智能卡读取器（<http://bit.ly/1fBYLhm>）。它使用PN532 NFC控制器，兼容很多开发库。

有一小部分的NFC库可用，各个库的完成情况不尽相同。一个最好的NFC工具库（<http://nfctools.org/>）是libnfc。libnfc是NFC的一个底层库，它支持控制协议ISO 14443A和-14443B（用于大多数的标签类型）、ISO-18092（P2P协议）和JIS-X-6319-4（FeliCa智能卡使用）。

该项目的libnfc路线图对于理解它是如何开发的相当有帮助。它可以在libnfc路线图页面（<http://bit.ly/libnfc-roadmap>）上找到。分层抽象模型是基于NFC的抽象模型，并留下了很大的空间以发展更高级的功能。

有一些基于libnfc的有用的子项目，包括libfreefare，它增加了读写Mifare标签的API。libfreefare可以在NFC工具网站上找到。还有libnfc的LLCP实现（<http://bit.ly/llcplibnfc>）。libfreefare对NFC论坛标签类型只提供了有限支持。它完全兼容Mifare Classic标签，但并不支持所有的4种NFC论坛标签类型。虽然在架构上能支持它们，但需要热心的程序员来开发并提交支持库。

libnfc是跨平台的，它能够运行在Windows、OS X和其他POSIX系统上，包括各种Linux版本。你会看到它在Raspberry Pi和BeagleBone上表现不错。

我们已经建了一个使用libnfc和libfreefare的Node.js库用来读写Mifare Classic标签。在这一点上，像libfreefare一样，它不提供对全部

NFC论坛标签类型的支持。还有，你需要安装libnfc和libfreefare，并多学习一点它们的知识。

通用功能

在安装任何NFC专用工具前，还有一些工作要做，磨刀不误砍柴工。

选择熟悉的编辑工具

你要将一些代码写到板子上，所以应该使用熟悉的编辑器。如果你喜欢用nano或vi，它们是基于Linux的编辑器，在Pi和BeagleBone上都有提供。如果你喜欢计算机上的图形编辑器，那么需要一种方法把文件从计算机传到嵌入式板子上。任何SFTP/FTP客户端都能胜任，或者也可以使用scp命令行方式来传文件。如果你的远程Web服务器上有写文件的工作流，那么在这里它也可用来传文件。

了解包管理器

运行Debian的Raspberry Pi和运行Ångström的BeagleBone Black有两个主要区别。第一个区别是，BeagleBone默认以administrator或root角色运行，而Pi的默认用户为pi。为了做管理任务，你需要有管理员权限。而以root角色运行时，你将自动获得这些权限。如果你没有以root角色运行，则可以使用sudo命令来获得管理权限。第一次使用sudo时，必须输入root密码，然后你就有管理员权限了。当你在一个命令前输入sudo时，系统会问你是否以管理员身份执行该命令（或超级用户，su do）。在本章中，你所看到的sudo命令，都假设这个命令必须以管理员身份运行。



更改默认密码，不管你是以root运行在BeagleBone Black上，还是以pi运行在Raspberry Pi上。保持密码唯一且安全是一个良好的习惯。

第二个区别是，它们使用不同的包管理器。Pi使用APT（先进的包工具），BeagleBone使用opkg包管理器。两者在许多方面是相似的，但也有一些工作使用不同的命令。我们把最常用的命令列在表9-1中。

表9-1：APT和opkg的常用功能

APT function	opkg function	Description
apt-get update	opkg update	Update list of available packages
apt-get upgrade	opkg upgrade	Upgrade all installed packages to the latest version
apt-cache search <pattern>	opkg search <pattern>	Full text search of available packages
apt-get install <package>	opkg install <package>	Install a package
apt-get remove <package>	opkg remove <package>	Remove a package
dpkg --get-selections	opkg list-installed	List installed packages

在使用任何包管理器之前，第一步是看看有没有最新的更新包。
在Raspberry Pi上，运行下面的命令：

```
$ sudo apt-get update
```

在BeagleBone Black上，运行下面的命令：

```
$ opkg update
```

设置日期和时间

确保板子上的日期和时间都是正确的。如果日期设置不正确，那么一些安装程序将无法正常工作。如果板子连上网络，Raspberry Pi会自动设置它的时钟，但BeagleBone则不会。要检查板子上的日期，输入date。要设置日期，请确保板子已连上网络，然后输入：

```
$ ntpdate -b -s -u pool.ntp.org
```

这是一个一次性的更新，将通过NTP协议 (<http://ntp.org>) 同步板子上的时钟与网络时间服务器上的时钟。Raspberry Pi将定期默认同步NTP服务器时间。BeagleBone则不会，请参阅Derek Molloy的网页来设置BeagleBone的时钟 (<http://bit.ly/1bEappq>) 以解决这个问题。要想使用ntpdate，pi用户必须先sudo apt-get install ntpdate。

建一个下载目录

接下来，在home目录下建一个downloads目录来存储下载的文件：

```
$ cd ~  
$ mkdir downloads  
$ cd downloads
```

然后在这个目录或其子目录中做安装的其他部分工作。

有几个工具可以用来编译和安装我们使用的大多数库，所以最好确认它们都在。当你使用apt-get（在Pi上）或opkg（在BeagleBone上）安装时，包管理器会通知你是否有最新包。对于Raspberry Pi，使用以下命令：

```
$ sudo apt-get install autoconf automake libtool libusb-1.0-0-dev
```

而对于BeagleBone Black，则使用：

```
$ opkg install autoconf automake libtool libusb-1.0-dev
```

BeagleBone特性

在BeagleBone上用HTTPS请求Git可能有些麻烦。为了避免这种情况，请确认板子上安装了适当的SSL证书，并且Git的配置如下。

你需要建立自己的Git配置文件来寻找SSL证书：

```
$ opkg install ca-certificates
```

如果得到回应告诉你已安装此包，那么OK。接下来，你需要告诉Git在哪里可以找到SSL证书：

```
$ git config --global http.sslCAinfo /etc/ssl/certs/ca-certificates.crt
```

Raspberry Pi特性

Raspberry Pi第一次设置安装时需要你有HDMI连接线和键盘来设置板子，但也可以避开。一种方式是打开一个终端连接通过串口连接到板子。你可以用3.3V USB转串口电缆，如USB到TTL串行电缆——调试/控制台电缆 (<http://www.adafruit.com/products/954>)，连接到串行端口。另一种方式是通过SSH连接到板子上。如果你不想购买电缆，还可以将Pi连接到局域网，并用下面的命令行找到它：

```
$ for i in 192.168.1.{1..254}; do ping -c 1 -W 3 $i & > /dev/null; done  
$ netstat -nr -f inet | grep -i "b8:27"
```

如果你的路由器没有192.168.1.1地址，那么更改上面命令行里地址的前3个字节为路由器地址的前3个字节的数值。

第一个命令会查询局域网中的所有设备。第二个命令会给你提供一个所有MAC地址前缀是B8: 27的设备列表。Pi板的MAC地址以这个前缀开始。如果在网络中找到你的Pi板子，请复制与它关联的IP地址，然后输入：

```
$ ssh pi@ip-address
```

当你得到一个登录提示时，输入用户名pi和默认密码raspberry。

在Raspberry Pi上安装Node.js

BeagleBone Black有预装的Node.js，但Pi没有，你需要使用以下步骤手动安装它。首先应确保你的Pi已接入Internet，然后输入：

```
$ cd ~/downloads  
$ curl -O http://nodejs.org/dist/v0.10.12/node-v0.10.12-linux-arm-pi.tar.gz ❶  
$ cd /usr/local  
$ sudo tar xzf ~/downloads/node-v0.10.12-linux-arm-pi.tar.gz ❷  
$ cd bin
```

```
$ sudo ln -s ./node-v0.10.12-linux-arm-pi/bin/node ③  
$ sudo ln -s ./node-v0.10.12-linux-arm-pi/bin/npm
```

①下载node。

②安装在/usr/local中。

③做符号链接（别名）到node和npm，以便可以在命令行简单地调用它们。

一旦为Pi安装好node，你就可以继续下一步了。

安装NFC工具

要安装本章项目的库，你需要确保板子已接入Internet。此外，你还需要一些其他软件库：

- libusb兼容库
- libnfc
- libfreefare
- ndef-mifare-classic-js
- ndef-js

这些库相互依赖，所以安装的顺序很重要。一旦它们都安装好了，你就可以读写NFC读取器了。在BeagleBone和Raspberry Pi上的安装细节相似，稍有不同。

既然你已经知道所用的NFC适配器是通过USB来连接的，那么不要惊讶在安装libnfc和libfreefare之前先要安装libusb。安装这些库，从libusb开始。

安装libusb兼容库

从libusb下载libusb-compat-0.1.5.tar.bz2 (<http://www.libusb.org/>) ,
解压缩并安装它, 像这样:

```
$ cd ~/downloads
$ wget http://sourceforge.net/projects/libusb/files/
    libusb-compat-0.1/libusb-compat-0.1.5/libusb-compat-0.1.5.tar.bz2
$ tar xjf libusb-compat-0.1.5.tar.bz2
$ cd libusb-compat-0.1.5
$ ./configure
$ make
$ make install ❶
```

❶Raspberry Pi用户需要使用sudo make install。

安装libnfc

接下来，你要下载和安装libnfc：

```
$ cd ~/downloads  
$ git clone https://code.google.com/p/libnfc/  
$ cd libnfc  
$ autoreconf -vis  
$ export PKG_CONFIG_PATH=/usr/local/lib/pkgconfig❶  
$ ./configure  
$ make  
$ make install❷
```

❶BeagleBone用户需要做这一步，但Pi用户不用；libfreefare的配置工具需要知道安装libnfc的地方。

❷Raspberry Pi用户需要使用sudo make install。

Raspberry Pi

将libnfc安装到/usr/local/lib目录；运行ldconfig将更新配置，以便在需要时链接器可以找到库：

```
$ sudo ldconfig
```

对于libnfc，还有一些安装后的配置工作要做。创建一个新的文件：

```
$ sudo nano /etc/modprobe.d/blacklist-libnfc.conf
```

将下面内容加到文件中：

```
blacklist pn533  
blacklist nfc
```

按“Ctrl+X”组合键，再按Y键保存并退出nano。然后从libnfc复制PN53x规则文件：

```
$ sudo cp contrib/udev/42-pn53x.rules /lib/udev/rules.d/
```

BeagleBone Black

与Pi类似，也有一些安装后的配置工作要做。创建一个新的文件：

```
$ nano /etc/modprobe.d/nfc.conf
```

将下面内容加到文件中：

```
install pn533 /bin/true  
install nfc /bin/true
```

按“Ctrl+X”组合键，然后按Y键保存并退出nano。

Raspberry Pi或BeagleBone安装测试

至此，板子上已经有了libnfc。插上NFC读取器，使用nfc-List命令进行测试，并产生如下输出：

```
$ nfc-list  
nfc-list uses libnfc libnfc-1.7.0-rc7-40-gbb5b712  
NFC device: SCM Micro / SCL3711-NFC&RW opened
```

如果你得到这样的结果，则说明读取器工作正常，可以继续下面的内容。

安装libfreefare

现在，你已经安装了libnfc，下一步下载并安装libfreefare：

```
$ cd ~/downloads  
$ git clone https://code.google.com/p/libfreefare/  
$ cd libfreefare  
$ autoreconf -vis  
$ ./configure  
$ make  
$ make install❶  
$ sudo ldconfig❷
```

❶Raspberry Pi用户需要使用sudo make install。

❷BeagleBone Black用户可以跳过这一步。

现在，你已经安装了所有必要的软件包，下面的示例对Raspberry Pi或BeagleBone的操作都相同，除非特别说明。

如果你对使用JavaScript或其他脚本语言来读写NDEF的应用程序感兴趣，那么可能不想知道太多关于libnfc和libfreefare的细节。它们提供了一个C/C++的API用于NFC读取器。下一节将介绍两者的一些亮点。接下来介绍一个Node.js库，用于从Mifare Classic标签读写NDEF，如果你只关心高层次的细节，则可跳过这部分。

libnfc和libfreefare的命令行工具

libnfc提供了一些命令行工具，你可以在utils/目录中找到。这些工具可以做一些底层NFC任务：

nfc-anticol	nfc-emulate-uid
nfc-read-forum-tag3	nfc-dep-initiator
nfc-list	nfc-relay
nfc-dep-target	nfc-mfclassic
nfc-relay-picc	nfc-emulate-forum-tag2
nfc-mfsetuid	nfc-scan-device
nfc-emulate-forum-tag4	nfc-multralight

最有趣的工具是nfc-mfclassic (<http://bit.ly/185A8Ij>)，它读取标签并把内容转存到一个文件中。它能获得标签内容的字节流，这个字节流不区分内容格式和元数据格式。插上NFC读取器，并把它靠近Mifare Classic标签。执行命令并产生如下输出：

```
$ nfc-mfclassic r b dump.mfd
NFC reader: SCM Micro / SCL3711-NFC&RW opened
Found MIFARE Classic card:
ISO/IEC 14443A (106 kbps) target:
    ATQA (SENS_RES): 00 04
    UID (NFCID1): 2d 10 18 07
    SAK (SEL_RES): 08
Guessing size: seems to be a 1024-byte card
Reading out 64 blocks |.....|
Done, 64 of 64 blocks read.
Writing data to file: dump.mfd ...Done.
```

nfc-classic的参数是：

r

读取受保护的Mifare Classic标签。

b

使用key b，即NDEF使用的Mifare加密方案。

dump.mfd

转存内容的输出文件。



SLC3711读取器离标签1cm距离时天线的响应最好。用一条USB延长线会方便读取器的使用。

所有的libnfc工具都是比较底层的工具。libfreefare提供了一些稍微高层次的工具。它能读取、写入和格式化Mifare Classic、Ultralight和DESFire标签。它也能读写这些标签内的NDEF消息。libfreefare的示例程序是用C语言编写的，可以在libfreefare目录下的examples/子目录中找到：

mifare-classic-format	mifare-desfire-ev1-configure-random-uid
mifare-classic-read-ndef	mifare-desfire-format
mifare-classic-write-ndef	mifare-desfire-info
mifare-desfire-access	mifare-desfire-read-ndef
mifare-desfire-create-ndef	mifare-desfire-write-ndef
mifare-desfire-ev1-configure-ats	mifare-ultralight-info

作为一个例子，下面是mifare-classic-format命令：

```
$ mifare-classic-format
Found Mifare Classic 1k with UID 6b57ee65. Format [yN] y
Formatting 16 sectors [...4...8...12...16] done.
```

可以用libfreefare来做的一件方便事情就是复制一个标签。要做到这一点，需要先读取一个标签，把内容转存到板子上的文件里，然后

将文件写入到另一个标签。下面是如何将标签内容写入到一个名为
ndef.bin的文件中：

```
$ mifare-classic-read-ndef -o ndef.bin
```

写入成功后，你可以使用xxd命令来查看。你能认出NDEF消息：

```
$ xxd ndef.bin
```

```
0000000: d101 3055 036d 2e66 6f75 7273 7175 6172 ..0U.m.foursquare
00000010: 652e 636f 6d2f 7665 6e75 652f 3461 3931 e.com/venue/4a91
00000020: 3735 3633 6639 3634 6135 3230 3430 3161 7563f964a520401a
00000030: 3230 6533 20e3
```

一旦得到了这个文件，你可以写到另一个标签中，像这样：

```
$ mifare-classic-write-ndef -i ndef.bin
```

在本章中libfreefare提供了最高层次的工具，即Node.js的mifare-classic-js和ndef-js包。

Node.js的NDEF读写

为了提供一个高层次的工具来读写NDEF消息，我们已经建立了两个Node.js包，即`ndef`和`mifare-classic`。使用这些包，你可以像之前章节中的例子那样写node web app。如果你使用的是BeagleBone，则也可以把这些包和BoneScript组合起来使用，BeagleBone的Arduino很像个能控制BeagleBone物理I/O的JavaScript框架。

你可以在GitHub上找到这两个包的完整源代码，在`https://github.com/don/ndef-js`上有NDEF消息库，在`https://github.com/don/mifare-classic-js`上有Mifare Classic标签库。它们也可以通过node包管理器npm来获取，下面的例子就是通过npm加载这两个包的。

使用NDEF包，你可以读写node上的NDEF消息。此包中的主要函数有：`encodeMessage()`，用于把NDEF消息编码成字节流写入标签；`decodeMessage()`，用于把传入的字节流解码成NDEF记录数组。它包含了大部分你已熟悉的PhoneGap-NFC插件里那些NDEF对象的功能，下面会有例子。

`mifare-classic`包为你提供了一个到libfreefare的JavaScript接口。因此，它只能工作在Mifare Classic标签，因为它依赖于libfreefare做读取操作。它有三个函数：`read()`和`write()`用于读写标签，`format()`用于将一个标签格式化为NDEF标签。

在`examples`/目录下，有这两个包的基本示例。只要你成功地安装了libnfc和libfreefare，那么安装和运行这些示例的步骤在BeagleBone、Raspberry Pi和其他基于Linux的板子上都是相同的。你创建项目文件后，可以使用npm来安装它们。

使用这两个包来读写和格式化一个NDEF格式的标签是相当简单的。首先，为应用程序新建一个目录，并在此目录下创建一个`package.json`文件：

```
$ cd ~
$ mkdir NodeNdefBasics
$ cd NodeNdefBasics
$ nano package.json
```

使用nano打开文件，把下面内容添加到文件中：

```
{
  "name": "NodeNdefBasics",
  "author": "Your Name",
  "version": "0.0.1",
  "description": "Read, write, or format a tag",
  "main": "read.js",
  "keywords": [
    "NDEF",
    "NFC"
  ],
  "dependencies": {
    "mifare-classic": "0.0.1",
    "ndef" : ">=0.0.4"
  },
  "engines": {
    "node": ">=0.8"
  }
}
```

按“Ctrl+X”组合键，再按Y键保存该文件。然后打开一个新文件read.js：

```
var ndef = require('ndef'),           // require ndef package
  mifare = require('mifare-classic'); // require mifare-classic package

mifare.read(function(error, buffer) {      // read tag
  if (error) {                          // if there's an error result
    console.log("Read failed ");        // let user know about the error
    console.log(error);
  } else {                             // you got an NDEF message
    // decode the message into a JSON object:
    var message = ndef.decodeMessage(buffer.toJSON());
    // print the message's records:
    console.log("Found NDEF message with " + message.length +
      // add "record" if there's only one, "records" if there's more:
      (message.length === 1 ? " record" : " records" ));
    // print the message:
    console.log(message);
```

```
        console.log(ndef.stringify(message));
    }
});
```

正如你所看到的，mifare-classic包的mifare.read（）函数设置了一个success回调函数作为参数。当它成功读取一个标签后，你就可以使用ndef.decodeMessage（）函数将缓冲数据转换成NDEF记录数组。

要安装ndef和mifare-classic包，请确保板子已经连接到Internet，然后输入：

```
$ npm install
```

一旦包安装成功，就可以把读取器放到离已写入NDEF消息的标签大概1cm距离的位置，然后输入：

```
$ node read.js
```

1s后，你应该得到这样的结果：

```
Found Mifare Classic 1k with UID bb60ee65.
NFC Forum application contains a "NDEF Message TLV".
Found NDEF message with 2 records
URI Record
http://nfc-tools.org
```

```
Text Record
Hello from nodejs
```

使用mifare-classic包格式化标签和用libfreefare格式化非常相似。这个包只是那个库的一个封装。创建一个新文件format.js：

```
var mifare = require('mifare-classic'); // require mifare-classic

mifare.format(function(error) { // format tag
  if (error) { // if there's an error,
    console.log("Format failed "); // report the error
    console.log(error);
  } else { // if the format works out OK
    console.log("Tag formatted OK"); // report that
```

```
    }  
});
```

安装好npm后，将读取器靠近要格式化的标签，并运行，你应该得到如下结果：

```
Found Mifare Classic 1k with UID 52d55be4.  
Formatting 16 sectors [...4...8...12...16] done.  
Tag formatted OK
```

格式化过程将需要几秒钟时间，如果在格式化完成之前拿开读取器，将会报错，所以最好保持不动。一旦格式化完成，你就可以准备写数据了。创建一个新文件write.js：

```
var ndef = require('ndef'),           // require ndef package  
  mifare = require('mifare-classic'), // require this package  
  message,                         // an NDEF message  
  bytes;                           // the bytes stream to write  
  
message = [  
  ndef.uriRecord("http://nfc-tools.org"), // make a URI record  
  ndef.textRecord("Hello from nodejs"), // make a text record  
  ndef.emptyRecord()                 // make an empty record  
];  
bytes = ndef.encodeMessage(message); // encode record as a byte stream  
mifare.write(bytes, function(error) { // write function  
  if (error) {                     // if there's an error,  
    console.log("Write failed "); // report the error  
    console.log(error);  
  } else {  
    console.log("Tag written successfully"); // report that tag was written  
  }  
});
```

最后三个示例的完整源代码可以在GitHub (<http://bit.ly/node-basics>) 上找到。

`mifare.write()` 函数需要两个参数，一个字节流和一个回调函数。如果成功，回调函数不返回任何东西；如果有错误，则返回错误信息。当你安装完并运行时，如果在有效范围内有个标签，那么你应该得到以下结果：

```
NDEF file is 45 bytes long.  
Found Mifare Classic 1k with UID 52d55be4.  
Tag written successfully
```

如果没有成功写入标签，你会得到这样的结果：

```
NDEF file is 45 bytes long.  
Found Mifare Classic 1k with UID 52d55be4.  
Write failed  
mifare-classic-write-ndef: No known authentication key for sector 0x0e  
mifare-classic-write-ndef: No known authentication key for sector 0x0a  
mad_write: Mifare Authentication Failed
```

对于读标签和格式化标签，成功的关键是将标签靠近读取器，并保持不动直到操作完成。如果你使用读取器示例读取标签，你会得到这样的输出：

```
Found Mifare Classic 1k with UID 52d55be4.  
NFC Forum application contains a "NDEF Message TLV".  
Found NDEF message with 3 records  
URI Record  
http://nfc-tools.org  
Text Record  
Hello from nodejs  
Empty Record
```

现在你明白了这两个node包的主要功能，是时候用它们做几个例子了。

标签写入器的Web界面

Node.js很容易做Web服务器用，因此有必要使用包为标签写入器创建一个Web界面。事实上，你可以用以前项目的index.html页面。

还是从新建目录和package.json文件开始：

```
$ cd ~  
$ mkdir NodeNdefWriterWeb  
$ cd NodeNdefWriterWeb  
$ nano package.json
```

这个包文件和写入器脚本文件之间的主要区别是依赖另外的express.js文件：

```
{  
  "name": "NodeNdefWriterWeb",  
  "author": "username",  
  "version": "0.0.2",  
  "description": "Write a tag from a browser",  
  "main": "index.js",  
  "keywords": [  
    "NDEF",  
    "NFC"  
,  
    "dependencies": {  
      "express": ">=3.0",  
      "mifare-classic": "0.0.1",  
      "ndef" : ">=0.0.4"  
    },  
    "engines": {  
      "node": ">=0.8"  
    }  
}
```

index.js和以前例子里的index.js非常相似，但也有一些不同，注意这个脚本文件的末尾：

```
/*
  NodeNdefWriterWeb.js
*/



var undef = require('ndef'),           // require ndef package 1
    mifare = require('mifare-classic'), // require this package
    express = require('express'),      // make an instance of express
    app = express(),                 // start Express framework
    server = require('http').createServer(app), // start an HTTP server
    record = {},                     // NDEF record to send
    deviceMessage = "",             // messages from writer device

app.use(express.bodyParser()); // use bodyParser middleware for express
server.listen(8080);          // listen for incoming requests on server
console.log("Listening for new clients on port 8080");

// respond to web GET requests with the index.html page:
app.get('*', function (request, response) {
  response.sendfile(__dirname + '/index.html');
});

// take anything that begins with /submit:
app.post('/submit', function (request, response) {
  var days = request.body.days,     // get the number of days from the body
      today = new Date(request.body.checkin), // get the time from the body
      // calculate the checkout timeStamp:
      departure = new Date(today.valueOf() + (days * 86400000)),
      nfcResponse,                // the response from the NFC reader
      message,                   // the NFC message to write
      bytes;                     // byte stream to write it with

  record.name = request.body.name;   // get the name from the body
  record.room = request.body.room;  // get the room number from the body
  // convert to unix time in seconds:
  record.checkin = Math.round(today.valueOf()/1000);
  record.checkout = Math.round(departure.valueOf()/1000);

  message = [
    undef.textRecord(JSON.stringify(record)), // make a text record 2
  ];
  bytes = undef.encodeMessage(message); // encode record as a byte stream

  mifare.write(bytes, function(error) { // write function 3
    if (error) {                  // if there's an error,
      nfcResponse = "Write failed"; // report it
      nfcResponse += error;
    }
  });
});
```

```
    } else {
      nfcResponse = "Tag written successfully" + JSON.stringify(record);
    }
    console.log(nfcResponse);// report that the tag was written
    // write the HTML head back to the browser:
    response.writeHead(200, {'Content-Type': 'text/html'});
    // send the data:
    response.write("Wrote the following to the card:<br>");
    response.write(nfcResponse + "<p>");
    // send the link back to the index and close the link:
    response.end("<a href=\"^\">Return to form</a>");
  });
}); // end of mifare.write()
}); // end of app.post()
```

①所添加的mifare-classic和ndef代替了串口。

②使用ndef包的助手创建一个文本记录，并将其编码为字节流。

③调用mifare.write（）写标签，并等待响应。

完整的源代码可以在GitHub (<http://bit.ly/node-writer-web>) 上找到。

这个例子的用户交互效果与前面我们提到的浏览器界面项目一致。在表单中输入数据，然后按提交按钮，同时把标签放在读取器上，NFC读取器就写标签了。如果写入成功，你应该能在浏览器中得到JSON字符格式的NDEF记录；如果失败，则会给出错误信息。

标签控制物理输出

使用嵌入式Linux板（像BeagleBone或Raspberry Pi）能做的一件很有趣的事情就是用它的通用I/O（GPIO）引脚来检测和控制物理世界。这些都是电路板的物理输入和输出引脚。第7章介绍的酒店门锁是一个很实用的应用，你也可以通过刷BeagleBone或Raspberry Pi来打开门锁，而不必使用门卡。

下面的例子显示了用mifare-classic和ndef-js包来控制电路板上的GPIO。电路板上的GPIO引脚带了几个LED指示灯，我们通过打开LED指示灯来表明从标签读取的NDEF消息中有多少条记录。

在BeagleBone上，将使用BoneScript包，它提供了一个API（在风格上与Arduino API相似）用以控制GPIO引脚。在Raspberry Pi上，将使用onoff npm包。

BeagleBone版本

在BeagleBone上，通用I/O（GPIO）引脚可用BoneScript控制，它是由Jason Kridner开发的node包，BeagleBoard产品的软件架构。BoneScript有很多命令类似于Arduino上的GPIO控制命令，如pinMode ()、digitalRead ()、digitalWrite ()等。它被内置在BeagleBone的Ångström版本里。事实上，如果打开网页浏览器，进入“<http://BeagleBone.local: 3000>”，而电路板在同一个局域网内，那么你将获得内置的BeagleBone Cloud9 IDE。通过它，你可以运行、停止和调试你写的任何BoneScript项目。

Cloud9 IDE把所有项目默认保存在/var/lib/cloud9/目录中。当你在Cloud9 IDE中编辑时，不需要担心package.json文件。这个IDE既方便又不方便，这意味着你不能轻易地include modules，如果modules不是IDE的native。当然，你也可以在自己的包里使用BoneScript，所以对于这个示例项目，可以跳过Cloud9 IDE，用你习惯的老方法来做。因为BoneScript被内置在系统里，所以不需要把它放在package.json文件中。node会自动使用本地版本。

这个示例和之前看到的标签读取器示例有所不同。还是从新建目录开始：

```
$ cd ~
$ mkdir NodeTagToLed
$ cd NodeTagToLed
$ nano package.json
```

下面是package.json文件：

```
{
  "name": "NodeTagToLed",
  "author": "username",
  "version": "0.0.1",
  "description": "Read a tag, light LEDs",
  "main": "index.js",
```

```

"keywords": [
  "NDEF",
  "NFC",
  "Bonescript"
],
"dependencies": {
  "mifare-classic": "0.0.1",
  "ndef" : ">=0.0.4"
  // note: Bonescript is built into the BB distribution.
  // if you include it in your package.json, it will cause errors
},
"engines": {
  "node": ">=0.8"
}
}

```

正如你所见，标签读取的变化不大，即使BoneScript关键字提到是可选的。index.js文件一开始也没啥变化，但是你要添加4个新的代码片段：

```

var ndef = require('ndef'),          // require ndef package
  mifare = require('mifare-classic'), // require this package
  io = require('bonescript');        // bonescript is built into the BB①□
  io.pinMode('USR0', 'out');        // set LED I/O pins as outputs ②
  io.pinMode('USR1', 'out');
  io.pinMode('USR2', 'out');
  io.pinMode('USR3', 'out');

setInterval(readTags, 2000);          // read every 2 seconds③
function readTags() {
  mifare.read(function(error, buffer) { // read tag
    if (error) {                      // if there's an error result
      console.log("Read failed ");    // let user know about the error
      console.log(error);
    } else {                          // you got an NDEF message
      // decode the message into a JSON object:
      var message = ndef.decodeMessage(buffer.toJSON());
      // print the message's records:
      console.log("Found NDEF message with " + message.length +
        // "record" if there's only one, "records" if there's more:
        (message.length === 1 ? " record" : " records"));
      // print the message:
      console.log(ndef.stringify(message));
      // loop over the LEDs and turn on one for each record: ④
      for (var pinNum=0; pinNum<4; pinNum++) {

```

```
var pin = 'USR' + pinNum;      // set pin name, USR0 - USR3
if (pinNum < message.length) { // USR0=1 record, USR1=2 records...
    io.digitalWrite(pin, 1);   // turn on pin
} else {
    io.digitalWrite(pin, 0);   // turn off pin
}
}
);
}
```

①此处将BoneScript文件引入程序。

②USR0 ~ USR3是BeagleBone Black上的4个内置LED指示灯。欲了解更多的有关GPIO引脚的命名知识，请打开浏览器，进入“<http://BeagleBone.local/Support/BoneScript/>”。参考资料就在板子上。

③为了保持脚本一直运行，设置间隔2s读取一次标签。把mifare.read () 封装到readTags () 函数里，后者会每2秒被调用一次。

④标签里NDEF消息的每条记录用于打开一个LED指示灯。首先连接引脚名称（USR0 ~ USR3），然后根据记录数打开或关闭引脚。

完整的源代码可以在GitHub (<http://bit.ly/node-tag-led>) 上找到。

使用npm把这个项目安装到电路板上，运行。它将一直运行，这样你就可以读取多个标签。当你读取一个标签时，它会计算记录数，并用来打开相应数量的LED指示灯。你写过的任何标签都能用来读。

Raspberry Pi版本

对于这个版本，你需要：

- 4个LED指示灯
- 4个220Ω电阻
- 1个小的无焊接面包板
- 4条母到公的跳线
- 4个头销

虽然BoneScript没有移植到Pi，但也有一些node库可以用来控制GPIO。虽然它们不像BoneScript那样功能全面，但也允许你打开和关闭GPIO引脚。下面是对BeagleBone例子的改动，但Pi使用onoff npm包。

为Pi新建目录。package.json是一样的，除了增加一个依赖关系：

```
"dependencies": {  
  "mifare-classic": "0.0.1",  
  "ndef" : ">=0.0.4"  
  "onoff" : ">=0.1.6"  
},
```

onoff包通过将虚拟文件写入系统目录来控制GPIO引脚。有关它的更多信息，请参见NPM的onoff页面(<http://npmjs.org/package/onoff>)。其实BoneScript在BeagleBone上也是同样操作的，所以这两个库很相似，示例中的代码也类似。

被标记的Pi的GPIO引脚显示如图9-1所示。

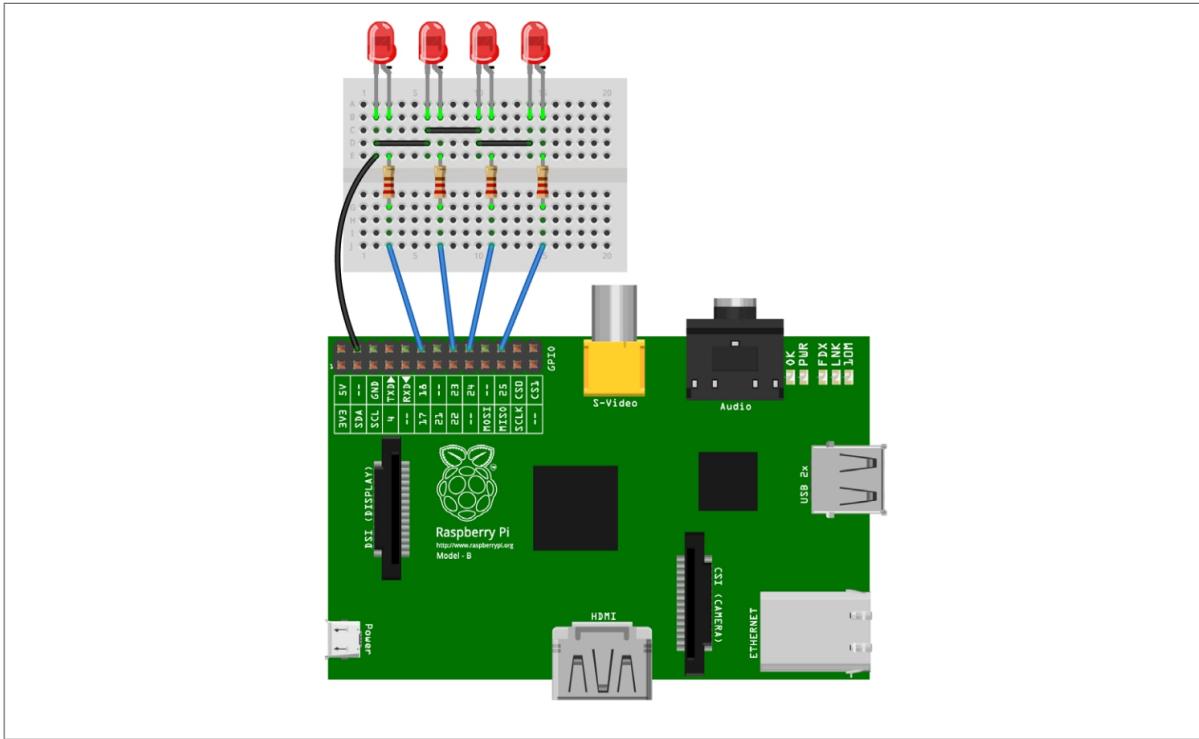


图9-1：Raspberry Pi上连接到GPIO引脚18、23、24和15的LED；电阻是 220Ω

为了使onoff在Pi上工作，你需要以管理员身份启用引脚。使用下面所示的一个简短脚本，来做到这一点：

```
var Gpio = require('onoff').Gpio,  
led = [];  
led[0] = new Gpio(18, 'out'), // set LED I/O pins as outputs  
led[1] = new Gpio(23, 'out'),  
led[2] = new Gpio(24, 'out'),  
led[4] = new Gpio(25, 'out');
```

保存为suexport.js文件，再用npm install安装整个项目。然后运行该脚本：

```
$ sudo node suexport.js
```

你不会看到任何输出，但该脚本会使下面的脚本以普通用户身份运行。

下面是Raspberry Pi版本的index.js文件，与BoneScript版本的差异已被标出：

```
var ndef = require('ndef'),           // require ndef package
    mifare = require('mifare-classic'), // require this package
    Gpio = require('onoff').Gpio,      // require onoff package
    led = [];                         // array of LEDs 1
led[0] = new Gpio(18, 'out'),         // set LED I/O pins as outputs 2
led[1] = new Gpio(23, 'out'),
led[2] = new Gpio(24, 'out'),
led[3] = new Gpio(25, 'out'),
setInterval(readTags, 2000);          // read every 2 seconds
function readTags() {
    mifare.read(function(error, buffer) { // read tag
        if (error) {                  // if there's an error result
            console.log("Read failed "); // let user know about the error
            console.log(error);
        } else {                      // you got an NDEF message
            // decode the message into a JSON object:
            var message = ndef.decodeMessage(buffer.toJSON());
            // print the message's records:
            console.log("Found NDEF message with " + message.length +
                // "record" if there's only one, "records" if there's more:
                (message.length === 1 ? " record" : " records" ));
            // print the message:
            console.log(ndef.stringify(message));
            // loop over the LEDs and turn on one for each record:
            for (var pinNum=0; pinNum<4; pinNum++) {
                console.log(pinNum);
                if (pinNum < message.length) { // LED0=1 record, LED1=2 records...
                    led[pinNum].writeSync(1); // turn on pin 3
                } else {
                    led[pinNum].writeSync(0); // turn off pin
                }
            }
        });
    }
}
```

1需要onoff代替bonescript。

2用数组表示4个LED引脚。为每个LED配一个GPIO对象。引脚数来自Pi的GPIO头。

③使用Gpio.writeSync () 代替digitalWrite () 。

安装并运行该脚本，运行结果和BeagleBone版本的一致：标签的NDEF消息里一条记录点亮一盏灯。

BeagleBone上的onoff包

这个项目也可以运行在BeagleBone上，只是有一些修改和注意事项。你无法控制内置的LED指示灯，但可以用onoff控制常规的GPIO引脚。你可以在网站 (<http://bit.ly/pin-headers>) 上找到BeagleBone上的GPIO引脚数。任何GPIO引脚都可用于输出到LED指示灯。

在撰写本文时，有一个semver问题，semver是npm的构建工具node-gyp所使用的版本检查工具。为了使用npm安装onoff的任何脚本，你需要编辑semver的相关配置文件：

```
$ nano /usr/lib/node_modules/npm/node_modules/node-gyp/lib/configure.js
```

查找以下行（可能在104~108行）：

```
if (semver.gte(version, '2.5.0') && semver.lt(version, '3.0.0')) {
  getNodeDir()
} else {
  failPythonVersion(version)
}
```

用下面这行替换上面4行，然后保存文件：

```
getNodeDir();
```

一旦改动，onoff将编译并安装在BeagleBone上。不过，你可能发现还是使用BoneScript更方便。

虽然这个例子比较简单，但它示范了通过NFC让嵌入式板和物理设备进行互操作的可能性。例如，第7章中的酒店门锁项目很容易被复制到Raspberry Pi或BeagleBone上。第6章中的灯光控制和音乐播放项目也可以用这些板子来做。欲了解更多的关于电路板控制GPIO的内容，请参阅前面提到的Matt Richardson的书，或者参阅Adafruit板子上优秀的教程。

嵌入式Linux上的NFC仍然是新生事物，还在发展中。libnfc作为进一步发展的基础，提供了巨大的潜力，我们希望看见它的进一步发展和子项目。特别是扩大其兼容的标签类型范围，像libllcp那样的P2P库交换如能引入将大大推进该技术在这些平台上的发展。嵌入式板上NFC驱动的物理应用的潜力是广阔的，本章所展示的平台提供了一种强大、价格低廉且简单的方式来开发这类应用程序。

总结

你已经看到了NFC在三个不同平台上的应用：Android用PhoneGap；Arduino用NDEF库；嵌入式Linux用libnfc。你已经了解了NDEF的结构和含义，也知道了各种NFC兼容标签，并通过NDEF进行了P2P交换。正如你所看到的，NFC技术提供了对RFID的一些有趣的改进，最显著的是独立于标签技术的数据格式，以及设备与设备之间不通过标签的通信能力。

当前的各种NFC API都还有改进空间。如果你是一个底层开发者，我们希望你能积极地参与，帮助拓展标签类型适用的范围，提供更高层次的库，并为各平台开发NDEF层次的API。如果你是高层次接口设计者，我们鼓励你在各种各样的应用中使用NFC。

附录A NFC规范代码

你可能经常使用一些NFC规范代码，它们在NFC论坛网站上的NFC规范（<http://bit.ly/nfc-tech-specs>）中也能找到，在此我们转载了一些以供参考。

NFC论坛规范列表提供了所有NFC规范的简短定义。当你很难记住缩略词，如LLCP、SNEP、TNF和RTD的意思时，可以到这个页面看看快速参考。

表A-1：类型名称格式 (TNF)

Type Name Format	Value
Empty	0x00
Well-Known type [NFC RTD]	0x01
MIME media-type [RFC 2046]	0x02
Absolute URI [RFC 3986]	0x03
External type [NFC RTD]	0x04
Unknown	0x05
Unchanged	0x06
Reserved	0x07

表A-2：常用的记录类型定义

Record Type	RTD code
Text	T
URI	U
Smart Poster	Sp
Alternative Carrier	ac
Handover Carrier	Hc
Handover Request	Hr
Handover Select	Hs

表A-3: URI识别码 (UIC)

Decimal	Hex	Protocol
0	0x00	None. The URI is added exactly as written.
1	0x01	http://www.
2	0x02	https://www.
3	0x03	http://
4	0x04	https://
5	0x05	tel:
6	0x06	mailto:
7	0x07	ftp://anonymous:anonymous@
8	0x08	ftp://ftp.
9	0x09	ftps:/
10	0x0A	sftp://
11	0x0B	smb://
12	0x0C	nfs://
13	0x0D	ftp://
14	0x0E	dav://
15	0x0F	news:
16	0x10	telnet://
17	0x11	imap:
18	0x12	rtsp://
19	0x13	urn:
20	0x14	pop:
21	0x15	sip:
22	0x16	sips:
23	0x17	tftp:
24	0x18	btspp://
25	0x19	btl2cap://
26	0x1A	btgoep://
27	0x1B	tcpobex://
28	0x1C	irdaobex://
29	0x1D	file://
30	0x1E	urn:epc:id:
31	0x1F	urn:epc:tag:
32	0x20	urn:epc:pat:

Decimal	Hex	Protocol
33	0x21	urn:epc:raw:
34	0x22	urn:epc:
35	0x23	urn:nfc:
36…255	0x24., 0xFF	保留为将来使用 (URI 将保存详细信息)

索引

A

- absolute URI TNF values , 51
- absolute URIs , 52
- active NFC, 14
- active RFID, 11, 13
- Adafruit, 19
- Alternative Carrier Records , 185
- Android
 - PhoneGap-NFC library and , 23
 - Tag Dispatch System , 89–93
 - Android Application Records (AAR) , 93, 176
 - Android Debug Bridge (ADB) , 105
 - Android Debug Monitor, 39
 - Android Developers Toolkit (ADT) , 24

on Linux, 25

on OS X, 25

on Windows, 25

Android software development kit (SDK), 24

installing, 24–29

Node.js, installing, 27

npm, installing, 27

platform tools, installing, 25–27

`AndroidManifest.xml`, 30

`ANDROID_HOME` path, 26

`ant.properties`, 30

AppLauncher NFC, 59

application design, 95–129

ADB shell, 104

background dispatch, enabling, 128

data formats, 101–104

global event handlers, 112

housekeeping functions, 110–112

PhoneGap Media API, 105

REST APIs and, 104

storage object for, 109

user interaction, 97–98

user interface, 106–109

user interface events, 118–126

Arduino, 131–171

browser interface for, 164

development environment, 133–140

Hotel Keycard project, 146–170

interaction, 148–149

libraries, installing, 138

microcontroller platform, 131

microcontrollers, 131–133

NDEF library, 140–146

NDEF reader device, 156–164

NDEF writer device, 150–156

peer-to-peer using, 192

radio controllers and, 133

reading NDEF in, 142–145

writing NDEF in, 145

Arduino IDE (Integrated Development Environment), 133–140

libraries, installing, 138
serial communications and, 137–138
assets, 31

We'd like to hear your suggestions for improving our indexes. Send email to index@oreilly.com.

B

background dispatch, enabling, 128
BeagleBone Black, 18
controlling physical output with, 214–217
editing workflow, 200
embedding in Linux, 199
Git and, 202
GPIO, 214–217
libnfc, installing, 205
libsub library, installing, 204
NFC tools, installing, 203–206
package managers for, 200
bin, 31
blocking functions, 145
Bluetooth vs.NFC, 12
browser interface
client-side code for, 165

for Arduino NDEF writer device , 164

server-side code , 166–170

build.xml , 31

C

card emulators , 14

CF (chunk flag) , 54

communication modes

NFC, 14

RFID, 13

compatible tags, 46

contacts, 46

Coordinated Universal Time (UTC) , 149

Cordova command-line interface (CLI) , 24, 28, 31

install on Linux, 28

install on OS X, 28

install on Windows, 28

PhoneGap vs., 29

D

damaged tags, 46
data types, 50
debugging
PhoneGap, 38–40
tools for, 38
delay function, 145
dependencies, 197
device-to-tag matching, 19
directory structure, 30
merges, 30
platforms, 30
plug-ins, 30
www, 30
dispatch systems, 75

E

embedded systems, 195–221
command line tools for, 206–208
GPIO and, 214–221
libfreefare, installing, 205
libnfc library, 204
libsub library, 204
NDEF, reading/writing in Node.js, 208–212
networks and, 196–199
on Linux, 196–199
package managers, 196–199
USB and, 196–199
web interfaces for, 212–214
workflow, editing, 200
empty TNF values, 51
environment variables, 26
event handlers

global, 112

NFC, 126–128

user interface, 118–126

event listeners (PhoneGap NFC), 75–77

events

global handlers, 112

multiple, listening for, 78–81

user interface, 118–126

external libraries, 164

external TNF values, 51

F

- filechoosers, 187
- filtering tags
 - by MIME type, 86–88
 - by record types, 84
- findstr, 39
- foreground dispatch systems, 75
- Foursquare check-in app
 - tag writer app, 61–72
 - using pre-built applications, 56–61

G

gen, 31

general purpose I/O (GPIO) , 214–221

global event handlers, 112

Google, 19

Google Wallet, 20, 193

grep, 39

H

- handover, 185–191
 - sending in PhoneGap, 186–191
 - static, 186
- Handover Carrier messages, 185
- Handover Request Record, 185
- Handover Selector Record, 185
- headers, 49
- Hello, World ! app, 24–34
- Android SDK, 24–29
- PhoneGap projects, creating, 29–32
- HomeSpot NFC-enabled Bluetooth receiver, 99
- Hotel Keycard project, 146–170
 - data format, 148–149
 - interaction, 148–149
- NDEF writer device, 150–156
- Hue system (Phillips), 98–104

data format for, 101–104

hub communications functions for, 114–118

REST API for, 104

I

- IL (ID length is present) , 54
- Indentive NFC, 19
- index.html, PhoneGap and, 33
- index.js, PhoneGap and, 33
- initiators, 13
- intent filters, 75, 89
- intents, types of, 89–93
- NDEF_DISCOVERED, 91
- TAG_DISCOVERED, 89
- TECH_DISCOVERED, 90
- Inter-Integrated Circuit communication (I2C) , 16, 138
- International Commission on Illumination (CIE) , 102
- International Standards Organization (ISO) , 13
- Internet Assigned Numbers Authority (IANA) ,

J

Japanese Industrial Standard (JIS) , 55

jshint, 38

L

libfreefare

installing, 205

usage, 206–208

libnfc library

installing, 204

testing, 205

usage, 206–208

libs, 31

libsub library, 204

Linux

ADT on, 25

CLI, installing, 28

embedded systems on, 196–199

embedding BeagleBone, 199

NFC on, 199

node.js, installing, 28

paths, creating, 26

Raspberry Pi, embedding, 199

listeners, 75–94

AAR and, 93

Android tag dispatch system and, 89–93

event, 75–77

for multiple events, 78–81

intents, types of, 89–93

NDEF messages, reading, 81–84

NDEF reader app, 77–88

tags, filtering, 84–88

local.properties, 31

locator app, 34–40

Logical Link Control Protocol (LLCP), 16, 173

M

- MB (message begin) , 53
- ME (message end) , 54
- memory in RFID tags , 13
- merges , 30
- message structure , 49–53
- message structure , 49–53
- payload , 53
- payload identifiers , 53
- payload types , 51–53
- type name format , 51
- URIs in , 52
- messages , 49
- microcontrollers , 131–133
- Mifare Classic , 16, 18
- Mifare Classic tag , 18
- Mifare Ultralight , 16

MIME media-type
in NDEF, 76

listener, 76

TNF values, 51

modes, 97

Mood Setter application, 95–129

ADB shell, 104

background dispatch, enabling, 128

housekeeping functions for, 110–112

hub communications functions, 114–118

Hue system (Phillips), 98–104

lighting event handlers, 118–126

music event handlers, 121–126

PhoneGap Media API, 105

user interaction, 97–98

N

- NDEF, 14, 49–73
 - browser interface for, 164
 - existing applications for, 56–61
 - helper functions, 71–72
 - library for Arduino, 140–146
 - messages, reading, 81–84
 - MIME media-types in, 76
 - Node.js, reading/writing in, 208–212
 - reader device, 156–164
 - reading in Arduino, 142–145
 - record layout, 53–56
 - tag writer app, 61–72
 - writer device, 150–156
 - writing in Arduino, 145
 - writing record to tag, 62–65
- NDEF formatable listener, 76

NDEF listener, 76

NDEF reader app, 77–88

NDEF records

creating new, 64

parameters, 64

payloads, 64

record ID, 64

record types, 64

signatures, 15

Smart Posters, 15

Type Name Format (TNF), 64

URIs, 15

`NdefMessage`, 142

`NdefRecord`, 142

`NDEF_DISCOVERED` intent, 91

near field communication (NFC)

active communication mode, 14

applications for, 20

architecture of, 15–19

Bluetooth vs., 12

communication modes, 14

defined, 12

event handlers, 126–128

event listeners, 76

NDEF and, 14

on Linux, 199

operation of, 14

passive communication mode, 14

RFID vs., 16

tag emulation mode, 193

tag types, 17

WiFi vs., 12

networks, 196–199

NFC (see near field communication)

NFC Connection Handover Specification, 185

NFC Data Exchange Format (see NDEF)

NFC operating modes, 14

card emulators, 14

peer-to-peer, 14

reader/writers, 14

NFC plug-in, installing, 41

NFC Reader project, 40–46

NFC reader app, writing, 42–45

tags, troubleshooting, 46

NFC TagWriter, 58

NFC Task Launcher, 57

records, 92

NFC-enabled devices, 46

NfcAdapter, 141

NfcTag, 142

Node.js, 27

Arduino and, 164

install on Linux, 28

install on OS X, 28

install on Windows, 27

installing on Raspberry Pi, 203

NDEF reading/writing, 208–212

npm, 27

O

operating modes for NFC devices , 14

P

package management systems, 197

package managers

date/time, setting, 201

downloads directory, creating, 202

for BeagleBone Black, 200

for Raspberry Pi, 200

package.json, 164

passive

NFC, 14

RFID, 11, 13

paths, creating, 26

on Linux, 26

on OS X, 26

on Vista, 26

on Windows, 26

on Windows 7, 26

on Windows 8.1, 26

on Windows XP, 26

payloads, 49, 52, 64

peer-to-peer exchange, 173–194

Arduino, using, 192

handover and, 185–191

handover messages in PhoneGap, 186–191

receiving, in PhoneGap, 183–185

sending, in PhoneGap, 175–182

tag emulation mode, 193

peer-to-peer mode, 14

Personal Health Device Communication (PHDC) Technical Specification, 20

Phillips Hue system, 95

PhoneGap, 23–47

Android SDK, installing, 24–29

application, 24–34

Cordova CLI for, 28

Cordova vs., 29

debugging, 38–40

event listeners , 75–77

Hello, World ! app , 24–34

helper functions , 71–72

index.html and , 33

index.js and , 33

locator app , 34–40

NFC plug-in , installing , 41

NFC Reader , 40–46

NFC reader app , writing , 42–45

projects , creating , 29–32

receiving peer-to-peer exchange , 183–185

sending handover messages in , 186–191

sending peer-to-peer exchange , 175–182

PhoneGap Media API , 105

PhoneGap-NFC library , 23–47 , 49–73

Android and , 23

helper functions , 71–72

platform tools , 26

platforms , 30

plug-ins , 30

POSIX time, 149

proguard-protect.txt, 31

project directories, 44

project repositories, 44

project.properties, 31

projects

Hello, World ! app, 24–34

Hotel Keycard, 146–170

locator app, 34–40

Mood Setter application, 95–129

NDEF reader app, 77–88

NFC Reader, 40–46

pre-built tag-writer app, 56–61

tag writer app, 61–72

R

radio controller, 133

radio frequency identification (RFID)

active, 11, 13

communication modes, 13

defined, 11

memory tags, 13

NFC vs., 16

operation of, 13

passive, 11, 13

standards for, 13

Raspberry Pi, 18

controlling physical output with, 217–221

editing workflow, 200

embedding in Linux, 199

Git and, 202

GPIO, 217–221

libnfc, installing, 205

libsub library, installing, 204

NFC tools, installing, 203–206

package managers for, 200

read mode, 97

reader/writers operating mode, 14

real-time clock (RTC) chip, 156

real-time operating systems (RTOS), 132

record ID, 64 record types, 52, 64

records (NDEF), 53–56

chunking, 56

filtering by type, 84

headers for, 54

size of, 55

types of, 66–70

writing to tags, 62–65

res, 31

reserved TNF values, 51

RESTful structures, 164

RFID (see radio frequency identification)

Samsung TecTiles, 18, 58

secure access modules, 193

secure elements, 193

serial communications, 137–138

Serial Peripheral Interface (SPI), 16, 138

Signature Record Type Definition, 156

signatures, 15

Simple NDEF Exchange Protocol (SNEP), 16, 173

simple text records, 15

smart card, 193

Smart Posters, 15

Sony FeLiCa RFID, 16

SparkFun, 19

Speed Studio, 19

SR (short record), 54

src, 31

standard inter-device serial protocols, 16

standards

for radio frequency identification (RFID), 13

inter-device serial protocols, 16

International Standards Organization (ISO) , 13

Japanese Industrial Standard (JIS) , 55

system variables , 26

T

tag discovered listener, 76

Tag Dispatch System (Android), 89–93

intents, types of, 89–93

tag emulation mode, 193

tag writer app, 61–72

web interface for, 212–214

TagAge, 19

tagPresent function, 145

tags, 17–19

controlling physical output with, 214–221

damaged, 46

filtering based on type, 84–88

matching to devices, 19

records, types of, 66–70

sources for, 18

troubleshooting, 46

types of, 17

TagStand Writer, 57

TAG_DISCOVERED intent, 89

targets, 13

TECH_DISCOVERED intent, 90

Tectile tags, 18

TNF values, 51

absolute URI, 51

empty, 51

external, 51

MIME media-type, 51

reserved, 51

unchanged, 51

unknown, 51

well-known, 51

tools, 26

Touch to Beam interface, 174

troubleshooting tags, 46

Type Name Format (TNF), 50, 64

U

- unique identifier numbers (UIDs) , 13
- Universal Asynchronous Receive-Transmit (UART) , 16, 138
- Unix time , 149
- unknown TNF values , 51
- URIs , 15, 52
- absolute , 52
- URLs , 52
- URNs , 52
- USB (Universal Serial Bus) , 16, 196–199
- user interaction , 97–98
- user interface , 106–109
- event handlers , 118–120
- user variables , 26

W

well-known TNF values, 51

WiFi vs.NFC, 12

workflow, editing, 200

write mode, 97

关于作者

Tom Igoe，在纽约大学Tisch艺术学院教物理计算和网络互动电讯项目的课程。在教学和研究过程中，他探索了利用数字技术来感知和响应更广泛的人类肢体语言的方法。他是*Making Things Talk and Getting Started with RFID*的作者。他和Dan O’Sullivan合作写了*Physical Computing: Sensing and Controlling the Physical World with Computers*。他是MAKE杂志的撰稿人，Arduino开源微控制器项目的合伙创始人。他希望有一天参观斯瓦尔巴特群岛和南极。

Don Coleman，一有都是个工程师，从机械工程师到软件工程师，现在再到硬件工程师，他在各学科间兜了一圈，成功跨越了各学科间的间隙。他是一个经验丰富的PhoneGap开发者，他自PhoneGap成立之初就开始使用它，并在全国做过关于使用PhoneGap的好处和优势的演讲。作为费城附近一家软件咨询公司Consulting for Chariot Solutions的负责人，他与团队和客户一起工作来重塑现有的技术，为未来奠定基础。

Brian Jepson，MAKE杂志的编辑，一个黑客，Providence Geeks and the Rhode Island Mini Maker Faire的组织者之一。他也是位于罗得岛州普罗维登斯市非营利性艺术中心AS220里的一个极客。AS220给了罗得岛的艺术家提供了未经审查的种子论坛，也提供了画廊、表演空间、制造设备、生活和工作的空间。

后记

本书封面上的动物是中美洲松鼠猴（*Saimiri oerstedii*）。这种小猴子具有鲜明的黑色和白色面罩，使它在巴拿马和哥斯达黎加的栖息地很容易被认出。松鼠猴是群居动物，一个群体有20~75只猴子。在猴类王国中，它们有最平等的社会结构。雌性并不形成统治结构，而在繁殖季节，雄性也只是变得好斗。

人们已经发现，中美洲松鼠猴群体中的雄性通常是有血缘关系的，所以它们显示出彼此明显的感情。这种行为，再加上群体内雄性平等的性交配权特征，是中美洲松鼠猴物种所特有的。相比之下，南美洲松鼠猴群体始终有一个严格的社会等级制度，雄性猴子为了性交配权而争夺猴王位置。

中美洲松鼠猴是杂食性动物，它们的食物包括昆虫、蜘蛛、水果、树叶、树皮、花和花蜜。它们有一种特别的方法来捕捉筑帐蝠。它们会通过寻找筑帐蝠的“帐篷”来寻找筑帐蝠。当它们找到筑帐蝠后，就会攀到更高的地方，从高处跳入帐篷，尝试吓晕筑帐蝠然后吃掉。考虑到其素食性，中美洲松鼠猴是一个非常重要的种子传播者和某些花的授粉者，包括西番莲。甚至有几种鸟类已经学会了跟随松鼠猴，希望在猴子冲出树木或灌木丛时能吃到额外的昆虫和小脊椎动物。

中美洲松鼠猴现有的数量，在哥斯达黎加估计为每平方英里36只，在巴拿马估计为每平方英里130只。据悉，森林砍伐、狩猎和捕获进行宠物贸易是它们越来越少的主要原因。栖息地的丧失，尤其是支离破碎的保护区，使它们难以建立大的种群。虽然猴子总量稍微好于40年前，但巴拿马仍不懈努力尽量扩大现有总量，特别是在该国的国家公园和野生动物保护区。

封面图片来自Riverside的Natural History。