一步步搭建物联网系统

其他:基于 REST 服务的最小物联网系统设计

一步步搭建物联网系统

0.1 前言

设计物联网系统是一种有意思的事情，我们需要考虑到软件、硬件、通讯等等不同的几个方案。探索不同的语言，不同的框架，形成不同的解决方案。

在文档中，我们将对设计物联网系统有一个简单的介绍，我们会探讨如何设计一个最小的物联网系统。

0.2 目标读者

本文档的目标读者是初入物联网领域，希望对物联网系统有一个大概的认识和把握，并学会如何掌握好一个基础的物联网系统的设计。

本文档对一些概念 (如) 只做了一些基本介绍，以及便于理解。如果想进一步了解这些概念，会列出一些推荐书目，以供参考。

- 硬件开发人员，对物联网有兴趣。
- 没有 web 开发经验
- 极少的 linux 使用经验
- 想快速将于生产环境

- 对硬件了解有限的开发人员。
- 没接触过 51、ARM、Arduino

- 想了解以下的东西
- RESTful 与 IOT
- CoAP 协议
- MQTT

0.3 不适合人群

- 丰富经验的开发者

0.4 介绍

关于内容的选择上，这是一个有意思的话题，我们很难判断不同的开发者用的是怎样的语言，用的是怎样的框架。

于是我们便自作主张地选择了那些适合于理论学习的语言、框架、硬件，去除掉其他那些我们不需要考虑的因素，如语法，复杂度等等。当然，这些语言、框架、硬件也是流行的，如果找到相关的文档。

- Arduino: 如果你从头开始学过硬件的话，那些你会爱上它的。
- Raspberry PI: 如果你从头编译过 GNU/Linux 的话，我想你会爱上她的。
- Python: 简单地来说，你可以方便地使用一些扩展，同时代码就表达了你的想法。
- PHP : 这是一门容易部署的语言，我想你只需要在你的 Ubuntu 机器上，执行一下脚本就能完成安装了。而且，如果你是一个硬件开发者的话，那么你会更容易找到其他开发者的。
- Javascript : 考虑到 CoAP、MQTT 等版本是基于 Nodejs 的话，而且这门语言已经无处不在了，而且会更加流行。
- HTML、CSS : 这是必须的，他们仍然也是无处不在。

0.4.1 为什么没有 C ?

如果你还想用 C 学理论的话，呵呵。

0.4.2 为什么不是 JAVA ?

大致有下面两个原因

- JAVA 学的人很多，然而不适合我们将主要精力集中于构建与学习，因为无关的代码太多了。
- 当时以及现在，我还是不喜欢 JAVA(ps: 更喜欢脚本语言，可以在更少的时候做更多的事)。

0.4.3 为什么没有 Android？

在 IOT 的 repo 中: https://github.com/gmszone/iot 是有 Android 的示例，然而这些理论不适合在这里讨论。

0.5 如何阅读

这是一个简单的建议，仅针对于在选择阅读没有经验的读者。

当前状态	
建议	
软件初学者	硬件开发者
从头阅读	从头阅读

有一天，走在回学校的路上，我在想：``未来是科技时代（现在也是），只是未来科技会无处不在，而如果我们对于周围的无处不在的代码一无所知的话，或许我们会成为黑客帝国中的一般人"。所以开始想着，人们会开始学习编程就像学习一门语言一样，直到有一天我看到了学习编程如同学习一门语言。这算是一个有趣的时间点，于是我开始想着像之前做最小物联网系统的那些步骤一样，写一个简单的入门。也可以补充好之前在这个最小物联网系统缺失的那些东西，给那些正在开始试图去解决编程问题的人。

我们先从身边的语言下手，也就是现在无处不在的 html+javascript+css。

1 无处不在的 HTML

从 html 开始的原因在于我们不需要去配置一个复杂的开发环境，也许你还不知道开始环境是什么东西，而这些需要去慢慢的了解才能接触，特别是对于普通的业余爱好者来说，对于专业的选手那些自然不是问题。HTML 是 Web 的核心语言，也算是基础的语言。

1.1 html 的 hello,world

Hello,world 是一个传统，所以在这里也遵循这个有趣的传统，我们所要做的事情事实很简单。虽然也有点 hack 的感觉，所以让我们新建一个文件叫 ``helloworld.html"。

(PS: 大部分人应该都是在 windows 下工作的，所以你需要新建一个文本，然后重命名，或者你需要一个编辑器，在这里推荐用 **sublime text**。破解不破解，注册不注册

都不会对你的使用有太多的影响。)

1. 新建文件
2. 输入

```
hello,world
```

3. 保存为 ->``helloworld.html",
4. 然后双击打开这个文件。正常情况下应该是刚好用你的默认浏览器打开。只要是一个现代的浏览器的话，应该可以看到上面显示的是 ``Hello,world"。

这才是最短的 hello,world 程序，其次呢？ruby 中的会是这样子的

```
2.0.0-p353 :001 > p "hello,world"
"hello,world"
    => "hello,world"
2.0.0-p353 :002 >
```

等等，如果你了解过 html 的话，会觉得这一点都不符合语法规则，但是他工作了，没有什么比安装完 Nginx 后看到 It works! 更让人激动了。

遗憾的是，它可能无法在所有的浏览器上工作，所以我们需要去调试其中的 bug。

1.1.1 调试 hello,world

我们会发现我们的代码在浏览器中变成了下面的代码，如果你和我一样用的是 chrome，那么你可以右键浏览器中的空白区域，点击审查元素，就会看到下面的代码。

```
<html>
    <head></head>
    <body>hello,world</body>
</html>
```

这个才是真正能在大部分浏览器上工作的代码，所以复制它到编辑器里吧。

1.1.2 说说 hello,world

我很不喜欢其中的 <*> 超文本标记语言

所以我们可以发现其中的关键词是标记 ------markup，也就是说 html 是一个 markup，head 是一个 markup，body 是一个 markup。

而后，我们真正工作的代码是在 body 里面，至于为什么是在里面，这个问题算是太复杂了。

1. 我们所学的汉语是别人创造的，我们所正在学的这门语言也是别人创造的。
2. 我们在自己的语言里遵循着他代表是个男的，她代替是个女的。

1.1.3 想用中文?

所以我们也可以把计算机语言与现实世界的语言划上一个等号。而我们所要学习的语言，因为没有一门真正意义上的汉语语言，所以我们便觉得这些很复杂，如果我们可以用汉语代换掉上面的代码的话

<语言>
<头><结束头>
<身体>你好，世界<结束身体>
<结束语言>

看上去很奇怪，只是因为是音译过去的原因，也许你会觉得这样会好理解一点，但是输入上可能一点儿也不方便，因为这键盘都不适合我们去输入汉字，也意味着可能你输入的会有问题。

让我们把上面的代码代替掉原来的代码然后保存，打开浏览器会看到下面的结果

<语言> <头><结束头> <身体>你好，世界<结束身体> <结束语言>

更不幸的结果可能是

<璇□▉> <澶□><缁撴潃澶□> <韬□絃>浣犲ソ锛屾笘鎺□<缁撴潃韬□絃> <缁撴潃璇□▉>

这是一个编码问题，对中文支持不友好。

所以我们把上面的代码改为和标记语言一样的结构

<语言>
    <头></头>
    <身体>你好，世界</身体>
<结束语言>

于是我们看到的结果便是

<语言> <头> <身体>你好，世界

被 chrome 浏览器解析成什么样了?

```
<html><head></head><body><语言>
        <头><!--头-->
        <身体> 你好，世界<!--身体-->
    <!--语言-->
</body></html>
```

以

结尾的是注释，写给人看的代码，不是给机器看的，所以机器不会去理解这些代码。

但是当我们把代码改成

```
<whatwewanttosay>你好世界</whatwewanttosay>
```

浏览器上面显示的内容就变成了

你好世界

或许你会觉得很神奇，但是这一点儿也不神奇，虽然我们的中文语法也遵循着标记语言的标准，但是我们的浏览器不支持中文标记。

结论:

1. 浏览器对中文支持不友好。
2. 浏览器对英文支持友好。

刚开始的时候不要对中文编程有太多的想法，这不是很现实的:

1. 现有的系统都是基于英语构建的，对中文支持不是很友好。

2. 中文输入的速度在某种程度上来说没有英语快。

我们离开话题已经很远了，但是这里说的都是针对于那些不满于英语的人来说的，只有当我们可以从头构建一个中文系统的时候才是可行的，这些包括的东西有 cpu，软件，硬件，而我们还需要考虑重新设计 cpu 的结构，在某种程度上来说会有些不现实。需要一代又一代的人的努力，只是在当前就更不现实了。忘记那些，师夷长之技以治夷。

1.2 其他 html 标记

添加一个标题，

```
<html>
    <head>
        <title>标 题</title>
    </head>
    <body>hello,world</body>
</html>
```

我们便可以在浏览器的最上方看到 ``标题" 二字，真实世界的淘宝网也包含了上面的东西，只是还包括了更多的东西，所以你也可以看懂那些我们可以看到的淘宝的标题。

```
<html>
<head>
    <title>标题</title>
</head>
<body>
hello,world
<h1>大标题</h1>
<h2>次标题</h2>
<h3>...</h3>
<ul>
    <li>列表 1</li>
    <li>列表 2</li>
</ul>
</body>
</html>
```

更多的东西可以在一些书籍上看到，这边所要说的只是一次简单的语言入门，其他的东西都和这些类似。

1.2.1 美妙之处

我们简单地上手了一门不算是语言的语言，浏览器简化了这其中的大部分过程，虽然没有 C 和其他语言来得有专业感，但是我们试着去开始写代码了。我们可能在未来的某一篇中可能会看到类似的语言，诸如 python，我们所要做的就是

```
$ python file.py
=>hello,world
```

然后在终端上返回结果。只是因为在我看来学会 html 是有意义的，简单的上手，而后再慢慢地深入，如果一开始我们就开始去理解指针，开始去理解类。我们甚至还知道程序是怎么编译运行的时候，在这个过程中又发生了什么。虽然现在我们也没能理解这其中发生了什么，但是至少展示了

1. 中文编程语言在当前意义不大，不现实，效率不高兼容性差
2. 语言的语法是固定的。（ps: 虽然我们也可以进行扩充，我们将会在后来支持上述的中文标记。）
3. 已经开始写代码，而不是还在配置开发环境。
4. 随身的工具才是最好的，最常用的 code 也才是实在的。

1.2.2 更多

我们还没有试着去解决，某商店里的糖一个 5 块钱，小明买了 3 个糖，小明一共花了多少钱的问题。也就是说我们学会的是一个还不能解决实际问题的语言，于是我们还需要学点东西如 javascript,css。我们可以理解为 Javascript 是解决问题的语言，html 是前端显示，css 是配置文件，这样的话，我们会在那之后学会成为一个近乎专业的程序员。我们刚学了下怎么在前端显示那些代码的行为，于是我们还需要 Javascript。

2 无处不在的 Javascript

Javascript 现在已经无处不在了，也许你正打开的某个网站他可能是 node.js+json +javascript+mustache.js 完成的，虽然你还没理解上面那些是什么，也正是因为你不理

解才需要去学习更多的东西。但是 Javascript 已经无处不在了，可能会在你手机上的某个 app 里，在你浏览的网页里，在你 IDE 中的某个进程中运行的。

2.1 Javascript 的 Hello,world

这里我们还需要有一个 helloworld.html，Javascript 是专为网页交互而设计的脚本语言，所以我们一点点来开始这部分的旅途，先写一个符合标准的 helloworld.html

```
<!DOCTYPE html>
<html>
    <head></head>
    <body></body>
</html>
```

然后开始融入我们的 javascript，向 HTML 中插入 Javascript 的方法，就需要用到 html 中的 <script> 标签，我们先用页面嵌入的方法来写 helloworld。

```
<!DOCTYPE html>
<html>
    <head>
        <script>
            document.write('hello,world');
        </script>
    </head>
    <body></body>
</html>
```

按照标准的写法，我们还需要声明这个脚本的类型

```
<!DOCTYPE html>
<html>
    <head>
        <script type="text/javascript">
            document.write('hello,world');
        </script>
    </head>
```

```
    <body></body>
</html>
```

没有显示 hello,world? 试试下面的代码

```
<!DOCTYPE html>
<html>
    <head>
        <script type="text/javascript">
            document.write('hello,world');
        </script>
    </head>
    <body>
        <noscript>
            disable Javascript
        </noscript>
    </body>
</html>
```

2.2 更 js 一点

我们需要让我们的代码看上去更像是 js，同时是以 js 结尾。C 语言的源码但是以 C 结尾的，所以我们要让我们的代码看上去更正式一点。于是我们需要在 helloworld.html 的同文件夹下创建一个 app.js，里面写着

```
document.write('hello,world');
```

同时我们的 helloworld.html 还需要告诉我们的浏览器 js 代码在哪里

```
<!DOCTYPE html>
<html>
    <head>
        <script type="text/javascript" src="app.js"></script>
    </head>
    <body>
        <noscript>
            disable Javascript
```

```
        </noscript>
    </body>
</html>
```

2.2.1 从数学出发

让我们回到第一章讲述的小明的问题，从实际问题下手编程，更容易学会编程。小学时代的数学题最喜欢这样子了 ------某商店里的糖一个 5 块钱，小明买了 3 个糖，小明一共花了多少钱的问题。在编程方面，也许我们还算是小学生。最直接的方法就是直接计算 3x5=?

```
document.write(3*5);
```

document.write 实际也我们可以理解为输出，也就是往页面里写入 3*5 的结果，在有双引号的情况下会输出字符串。我们便会在浏览器上看到 15，这便是一个好的开始，也是一个不好的开始。

2.3 设计和编程

对于我们的实际问题如果总是止于所要的结果，很多年之后，我们成为了 code monkey。对这个问题进行一次设计，所谓的设计有些时候会把简单的问题复杂化，有些时候会使以后的扩展更加简单。这一天因为这家商店的糖价格太高了，于是店长将价格降为了 4 块钱。

```
document.write(3*4);
```

于是我们又得到了我们的结果，但是下次我们看到这些代码的时候没有分清楚哪个是糖的数量，哪个是价格，于是我们重新设计了程序

```
tang=4;
num=3;
document.write(tang*num);
```

这才能叫得上是程序设计，或许你注意到了 ``;'' 这个符号的存在，我想说的是这是另外一个标准，我们不得不去遵守，也不得不去 fuck。

2.3.1 函数

记得刚开始学三角函数的时候，我们会写

```
sin 30=0.5
```

而我们的函数也是类似于此，换句话说，因为很多搞计算机的先驱都学好了数学，都把数学世界的规律带到了计算机世界，所以我们的函数也是类似于此，让我们做一个简单的开始。

```
function hello(){
    return document.write("hello,world");
}
hello();
```

当我第一次看到函数的时候，有些小激动终于出现了。我们写了一个叫 hello 的函数，它返回了往页面中写入 hello,world 的方法，然后我们调用了 hello 这个函数，于是页面上有了 hello,world。

```
function sin(degree){
    return document.write(Math.sin(degree));
}
sin(30);
```

在这里 degree 称之为变量，也就是可以改变的量。于是输出了 -0.9880316240928602，而不是 0.5，因为这里用的是弧度制，而不是角度制。

```
sin(30)
```

的输出结果有点类似于 sin 30。写括号的目的在于，括号是为了方便解析，这个在不同的语言中可能是不一样的，比如在 ruby 中我们可以直接用类似于数学中的表达:

```
2.0.0-p353 :004 > Math.sin 30
=> -0.9880316240928618
2.0.0-p353 :005 >
```

我们可以在函数中传入多个变量，于是我们再回到小明的问题，就会这样去写代码。

```
function calc(tang,num){
    result=tang*num;
    document.write(result);
}
calc(3,4);
```

但是从某种程度上来说，我们的 calc 做了计算的事又做了输出的事，总的来说设计上有些不好。

2.3.2 重新设计

我们将输出的工作移到函数的外面，

```
function calc(tang,num){
    return tang*num;
}
document.write(calc(3,4));
```

接着我们用一种更有意思的方法来写这个问题的解决方案

```
function calc(tang,num){
    return tang*num;
}
function printResult(tang,num){
    document.write(calc(tang,num));
}
printResult(3, 4)
```

看上去更专业了一点点，如果我们只需要计算的时候我们只需要调用 calc，如果我们需要输出的时候我们就调用 printResult 的方法。

2.3.3 object 和函数

我们还没有说清楚之前我们遇到过的 document.write 以及 Math.sin 的语法看上去很奇怪，所以让我们看看他们到底是什么，修改 app.js 为以及内容

```
document.write(typeof document);
document.write(typeof Math);
```

typeof document 会返回 document 的数据类型，就会发现输出的结果是

```
object object
```

所以我们需要去弄清楚什么是 object。对象的定义是

> 无序属性的集合，其属性可以包含基本值、对象或者函数。

创建一个 object，然后观察这便是我们接下来要做的

```
store={};
store.tang=4;
store.num=3;
document.write(store.tang*store.num);
```

我们就有了和 document.write 一样的用法，这也是对象的美妙之处，只是这里的对象只是包含着基本值，因为

```
typeof story.tang="number"
```

一个包含对象的对象应该是这样子的。

```
store={};
store.tang=4;
store.num=3;
document.writeln(store.tang*store.num);

var wall=new Object();
wall.store=store;
document.write(typeof wall.store);
```

而我们用到的 document.write 和上面用到的 document.writeln 都是属于这个无序属性集合中的函数。

下面代码说的就是这个无序属性集中中的函数。

```
var IO=new Object();
function print(result){
```

```
    document.write(result);
};
IO.print=print;
IO.print("a obejct with function");
IO.print(typeof IO.print);
```

我们定义了一个叫 IO 的对象，声明对象可以用

```
var store={};
```

又或者是

```
var store=new Object{};
```

两者是等价的，但是用后者的可读性会更好一点，我们定义了一个叫 print 的函数，他的作用也就是 document.write，IO 中的 print 函数是等价于 print() 函数，这也就是对象和函数之间的一些区别，对象可以包含函数，对象是无序属性的集合，其属性可以包含基本值、对象或者函数。

复杂一点的对象应该是下面这样的一种情况。

```
var Person={name:"phodal",weight:50,height:166};
function dream(){
    future;
};
Person.future=dream;
document.write(typeof Person);
document.write(Person.future);
```

而这些会在我们未来的实际编编程中用得更多。

2.3.4 面向对象

开始之前先我们简化上面的代码，

```
Person.future=function dream(){
    future;
}
```

看上去比上面的简单多了，不过我们还可以简化为下面的代码。。。

```
var Person=function(){
    this.name="phodal";
    this.weight=50;
    this.height=166;
    this.future=function dream(){
        return "future";
    };
};
var person=new Person();
document.write(person.name+"<br>");
document.write(typeof person+"<br>");
document.write(typeof person.future+"<br>");
document.write(person.future()+"<br>");
```

只是在这个时候 Person 是一个函数，但是我们声明的 person 却变成了一个对象一个 **Javascript** 函数也是一个对象，并且，所有的对象从技术上讲也只不过是函数。这里的 +``
" 是 HTML 中的元素，称之为 DOM，在这里起的是换行的作用，我们会在稍后介绍它，这里我们先关心下 this。this 关键字表示函数的所有者或作用域，也就是这里的 Person。

上面的方法显得有点不可取，换句话说和一开始的

```
document.write(3*4);
```

一样，不具有灵活性，因此在我们完成功能之后，我们需要对其进行优化，这就是程序设计的真谛 ------解决完实际问题后，我们需要开始真正的设计，而不是解决问题时的编程。

```
var Person=function(name,weight,height){
    this.name=name;
    this.weight=weight;
    this.height=height;
    this.future=function(){
        return "future";
    };
```

```
};
var phodal=new Person("phodal",50,166);
document.write(phodal.name+"<br>");
document.write(phodal.weight+"<br>");
document.write(phodal.height+"<br>");
document.write(phodal.future()+"<br>");
```

于是，产生了这样一个可重用的 Javascript 对象,this 关键字确立了属性的所有者。

2.4 其他

Javascript 还有一个很强大的特性，也就是原型继承，不过这里我们先不考虑这些部分，用尽量少的代码及关键字来实际我们所要表达的核心功能，这才是这里的核心，其他的东西我们可以从其他书本上学到。

所谓的继承，

```
var Chinese=function(){
    this.country="China";
}

var Person=function(name,weight,height){
    this.name=name;
    this.weight=weight;
    this.height=height;
    this.futrue=function(){
        return "future";
    }
}
Chinese.prototype=new Person();

var phodal=new Chinese("phodal",50,166);
document.write(phodal.country);
```

完整的 Javascript 应该由下列三个部分组成:

- 核心 (ECMAScript)------核心语言功能

- 文档对象模型 (DOM)------访问和操作网页内容的方法和接口
- 浏览器对象模型 (BOM)------与浏览器交互的方法和接口

我们在上面讲的都是 ECMAScript，也就是语法相关的，但是 JS 真正强大的，或者说我们最需要的可能就是对 DOM 的操作，这也就是为什么 jQuery 等库可以流行的原因之一，而核心语言功能才是真正在哪里都适用的，至于 BOM 真正用到的机会很少，因为没有好的统一的标准。

一个简单的 DOM 示例,

```
<!DOCTYPE html>
<html>
<head>
</head>
<body>
    <noscript>
        disable Javascript
    </noscript>
    <p id="para" style="color:red">Red</p>
</body>
    <script type="text/javascript" src="app.js"></script>
</html>
```

我们需要修改一下 helloworld.html 添加

```
<p id="para" style="color:red">Red</p>
```

同时还需要将 script 标签移到 body 下面，如果没有意外的话我们会看到页面上用红色的字体显示 Red，修改 app.js。

```
var para=document.getElementById("para");
para.style.color="blue";
```

接着，字体就变成了蓝色，有了 DOM 我们就可以对页面进行操作，可以说我们看到的绝大部分的页面效果都是通过 DOM 操作实现的。

2.5 美妙之处

这里说到的 Javascript 仅仅只是其中的一小小部分，忽略掉的东西很多，只关心的是如何去设计一个实用的 app，作为一门编程语言，他还有其他强大的内制函数，要学好需要一本有价值的参考书。这里提到的只是其中的不到 20% 的东西，其他的 80% 或者更多会在你解决问题的时候出现。

- 我们可以创建一个对象或者说函数，它可以包含基本值、对象或者函数。
- 我们可以用 Javascript 修改页面的属性，虽然只是简单的示例。
- 我们还可以去解决实际的编程问题。

3 无处不在的 CSS

CSS 或许你觉得他一点儿也不重要，HTML 好比是建筑的框架，CSS 就是用于装修房子。那么 Javascript 呢，我听到的最有趣的说法是小三，先让我们回到代码上来吧。

3.1 CSS

下面就是我们之前说到的代码，css 将 Red 三个字母变成了红色。

```
<!DOCTYPE html>
<html>
<head>
</head>
<body>
    <p id="para" style="color:red">Red</p>
</body>
    <script type="text/javascript" src="app.js"></script>
</html>
```

只是，

```
var para=document.getElementById("para");
para.style.color="blue";
```

将字体变成了蓝色，CSS+HTML 让页面有序的工作着，但是 Javascript 打乱了这些秩序，不过却也让生活多姿多彩，小三不都是这样的么 ------终于可以理解，为什么

以前人们对于 Javascript 没有好感了？不过这里要讲的是正室，也就是 CSS，这时还没有 Javascript。

Red

Alt text

3.2 关于 CSS

这不是一篇好的关于讲述 CSS 的书籍，所以不会去说 CSS 是怎么来的，有些东西既然我们可以很容易从其他地方知道，也就不需要花太多时间去重复。诸如重构等这些的目的之一也在于去除重复的代码，不过有些重复是不可少的，也是有必要的，而通常这些东西可能是由其他地方复制过来的。

到目前为止我们没有依赖于任何特殊的硬件或者是软件，对于我们来说我们最基本的需求就是一台电脑，或者可以是你的平板电脑，当然也可以是你的智能手机，因为他们都有个浏览器，而这些都是能用的，对于我们的 CSS 来说也不会有例外的。

CSS 是来自于 (Cascading Style Sheets)，到今天我也没有记得他的全称，CSS 还有一个中文名字是层叠式样式表，翻译成什么样的可能并不是我们关心的内容，我们需要关心的是他能做些什么。作为三剑客之一，它的主要目的在于可以让我们方便灵活地去控制 Web 页面的外观表现。我们可以用它做出像淘宝一样复杂的界面，也可以像我们的书本一样简单，不过如果要和我们书本一样简单的话，可能不需要用到 CSS。HTML 一开始就是依照报纸的格式而设计的，我们还可以继续用上面说到的编辑器，又或者是其他的。如果你喜欢 DreamWeaver 那也不错，不过一开始使用 IDE 可无助于我们写出良好的代码。

忘说了，CSS 也是有版本的，和 windows，Linux 内核等等一样，但是更新可能没有那么频繁，HTML 也是有版本的，JS 也是有版本的，复杂的东西不是当前考虑的内容。

3.3 代码结构

对于我们的上面的 Red 示例来说，如果没有一个好的结构，那么以后可能就是这样子。

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

```
<head>
</head>
<body>
   <p style="font-size: 22px;color:#f00;text-align: center;padding-
left: 20px;">如果没有一个好的结构</p>
   <p style=" font-size:44px;color:#3ed;text-indent: 2em;padding-
left: 2em;">那么以后可能就是这样子。。。。</p>
</body>
</html>
```

虽然我们看到的还是一样的:

如果没有一个好的结构

那么以后可能就是这样子。。。。

Alt text

于是我们就按各种书上的建议重新写了上面的代码

```
<!DOCTYPE html>
<html>
<head>
    <title>CSS example</title>
    <style type="text/css">
        .para{
            font-size: 22px;
            color:#f00;
            text-align: center;
            padding-left: 20px;
        }
        .para2{
            font-size:44px;
            color:#3ed;
            text-indent: 2em;
            padding-left: 2em;
```

```
        }
    </style>
</head>
<body>
    <p class="para">如果没有一个好的结构</p>
    <p class="para2">那么以后可能就是这样子。。。。</p>
</body>
</html>
```

总算比上面好看也好理解多了，这只是临时的用法，当文件太大的时候，正式一点的写法应该是下面:

```
<!DOCTYPE html>
<html>
<head>
    <title>CSS example</title>
    <style type="text/css" href="style.css"></style>
</head>
<body>
    <p class="para">如果没有一个好的结构</p>
    <p class="para2">那么以后可能就是这样子。。。。</p>
</body>
</html>
```

我们需要

```
<!DOCTYPE html>
<html>
<head>
    <title>CSS example</title>
    <link href="./style.css" rel="stylesheet" type="text/css" />
</head>
<body>
    <p class="para">如果没有一个好的结构</p>
    <p class="para2">那么以后可能就是这样子。。。。</p>
</body>
</html>
```

然后我们有一个像 app.js 一样的 style.css 放在同目录下，而他的内容便是

```
.para{
    font-size: 22px;
    color:#f00;
    text-align: center;
    padding-left: 20px;
}
.para2{
    font-size:44px;
    color:#3ed;
    text-indent: 2em;
    padding-left: 2em;
}
```

这代码和 JS 的代码有如此多的相似

```
var para={
    font_size:'22px',
    color:'#f00',
    text_align:'center',
    padding_left:'20px',
}
```

而 22px、20px 以及 #f00 都是数值，因此。。

```
var para={
    font_size:22px,
    color:#f00,
    text_align:center,
    padding_left:20px,
}
```

目测差距已经尽可能的小了，至于这些话题会在以后讨论到，如果要让我们的编译器更正确的工作，那么我们就需要非常多的这种符号，除非你乐意去理解:

```
(dotimes (i 4) (print i))
```

总的来说我们减少了符号的使用，但是用 lisp 便带入了更多的括号，不过这是一种简洁的表达方式，也许我们可以在其他语言中看到，或者说用这个去。。

```
\d{2}/[A-Z][a-z][a-z]/\d{4}
```

没有什么会比一开始不理解那是正则表达式，然后去修改上面的代码，为的是去从一堆数据中找出某日/某月/某年。

这门语言可能是为设计师而设计的，但是设计师大部分还是不懂编程的，不过相对来说还是比其他语言好理解一些。

3.4 样式与目标

下面也就是我们的样式

```
.para{
    font-size: 22px;
    color:#f00;
    text-align: center;
    padding-left: 20px;
}
```

我们的目标就是

如果没有一个好的结构

所以样式和目标在这里牵手了，问题是他们是如何在一起的呢？下面就是 CSS 与 HTML 沟通的重点所在了:

3.4.1 选择器

我们用到的选择器叫做类选择器，也就是 class，或者说应该称之为 class 选择器更合适。与类选择器最常一起出现的是 ID 选择器，不过这个适用于比较高级的场合，诸如用 JS 控制 DOM 的时候就需要用到 ID 选择器。而基本的选择器就是如下面的例子:

```
p.para{
    color:#f0f;
}
```

将代码添加到 style.css 的最下面会发现 ``如果没有一个的结构" 变成了粉红色，当然我们还会有这样的写法

```
p>.para{
    color:#f0f;
}
```

为了产生上面的特殊的样式，虽然不好看，但是我们终于理解什么叫层叠样式了，下面的代码的重要度比上面高，也因此有更高的优先规则。

而通常我们可以通过一个

```
p{
    text-align:left;
}
```

这样的元素选择器来给予所有的 p 元素一个左对齐。

还有复杂一点的复合型选择器，下面的是 HTML 文件

```
<!DOCTYPE html>
<html>
<head>
    <title>CSS example</title>
    <link href="./style.css" rel="stylesheet" type="text/css" />
</head>
<body>
    <p class="para">如果没有一个好的结构</p>
    <div id="content">
        <p class="para2">那么以后可能就是这样子。。。。</p>
    </div>
</body>
</html>
```

还有 CSS 文件

```
.para{
    font-size: 22px;
```

```
    color:#f00;
    text-align: center;
    padding-left: 20px;
}
.para2{
    font-size:44px;
    color:#3ed;
    text-indent: 2em;
    padding-left: 2em;
}

p.para{
    color:#f0f;
}
div#content p {
    font-size:22px;
}
```

3.5 更有趣的 CSS

一个包含了 para2 以及 para_bg 的例子

```
    <div id="content">
       <p class="para2 para_bg">那么以后可能就是这样子。。。。</
p>
    </div>
```

我们只是添加了一个黑色的背景

```
.para_bg{
    background-color:#000;
}
```

重新改变后的网页变得比原来有趣了很多，所谓的继承与合并就是如上面的例子。

我们还可以用 CSS3 做出有趣的效果，而这些并不在我们的讨论范围里面，因为我们讨论的是 be a geek。

或许我们写的代码都是那么的简单，从 HTML 到 Javascript，还有现在的 CSS，只是有一些东西才是核心的，而不是去考虑一些基础的语法，基础的东西我们可以从实践的过程中一一发现。但是我们可能发现不了，或者在平时的使用中考虑不到一些有趣的用法或者说特殊的用法，这些可以从观察一些比较好的设计的代码中学习到。复杂的东西可以变得很简单，简单的东西也可以变得很复杂。

4 无处不在的三剑客

这时我们终于了解了我们的三剑客，就这么可以结合到一起了，HTML+Javascript +CSS 是这一切的基础。而我们用到的其他语言如 PHP、Python、Ruby 等等的最后都会变成上面的结果，当然还有 Coffeescript 之类的语言都是以此为基础，这才是我们需要的知识。

4.1 Hello,Geek

有了一些些的基础之后，我们终于能试着去写一些程序了。也是时候去创建一个像样的东西，或许你在一些见面设计方面的书籍看过类似的东西，可能我写得也没有那些内容好，只是这些都是一些过程。过去我们都是一点点慢慢过来的，只是现在我们也是如此，技术上的一些东西，事实上大家都是知道的。就好比我们都觉得我们可以开个超市，但是如果让我们去开超市的话，我们并不一定能赚钱。

学习编程的目的可能不在于我们能找到一份工作，那只是在编程之外的东西，虽然确实也是很确定的。但是除些之处，有些东西也是很重要的。、

过去没有理解为什么会一些人会不厌其烦地去回答别人的问题，有时候可能会想是一种能力越大责任越大的感觉，但是有时候在写一些博客或者回答别人的问题的时候我们又重新思考了这些问题，又重新学习了这些技能。所以这里可能说的不是关于编程的东西而是一些编程以外的东西，关于学习或者学习以外的东西。

4.2 从源码学习

过去总会觉得学了一种语言的语法便算是学了一种语言，于是有一天发现到了这个语言的项目上的时候，虽然会写上几行代码，但是却不像这语言的风格。于是这也是这一篇的意义所在了：

4.2.1 HTML

写好 HTML 的一个要点在于看别人写的代码，这只是一方面，我们所说的 HTML 方面的内容可能不够多，原因有很多，很多东西都需要在实战中去解决。读万卷书和行万里路，分不清哪个有重要的意义，但是如果可以同时做好两个的话，成长会很快的。

写好 HTML 应该会有下面的要点

- 了解标准及遵守绝大多数标准
- 注重可读性，从 ID 及 CLASS 的命名
- 关注 SEO 与代码的联系

或许在这方面我也算不上很了解，不过按笔者的经验来说，大致就是如此。

多数情况下我们的 HTML 是类似于下面这样子的

```
<div class="col-md-3 right">
    {% nevercache %}
    {% include "includes/user_panel.html" %}
    {% endnevercache %}
    <div class="panel panel-default">
    <div class="panel-body">
    {% block right_panel %}
    {% ifinstalled mezzanine.twitter %}
    {% include "twitter/tweets.html" %}
    {% endifinstalled %}
    {% endblock %}
    </div>
    </div>
</div>
```

换句话说 HTML 只是基础，而不是日常用到的。我们的 HTML 是由 template 生成的，我们可以借助于 mustache.js 又或者是 angluarjs 之类的 js 库来生成最后的 HTML，所以这里只是一个开始。

还需要了解的一部分就是 HTML 的另外一个重要的部分，DOM 树形结构

4.3 DOM 树形结构图

4.3.1 javascript

这里以未压缩的 jQuery 源码和 zepto.js 作一个小小的比较，zepto.js 是兼容 jQuery 的，因此我们举几个有意思的函数作一简单的比较，关于源码可以在官网上下载到。

在 zepto.js 下面判断一个值是否是函数的方面如下，

```
function isFunction(value) { return type(value) == "function" }
```

而在 jQuery 下面则是这样的

```
isFunction: function( obj ) {
    return jQuery.type(obj) === "function";
},
```

而他们的用法是一样的，都是

```
$.isFunction();
```

jQuery 的作法是将诸如 isFunction,isArray 这些函数打包到 jQuery.extend 中，而 zepto.js 的则是也是这样的，只不过多了一行

```
$.isFunction = isFunction
```

遗憾的是我也没去了解过为什么，之前我也没有看过这些库的代码，所以这个问题就要交给读者去解决了。jQuery 里面提供了函数式编程接口，不过 jQuery 更多的是构建于 CSS 选择器之上，对于 DOM 的操作比 javascript 自身提供的功能强大得多。如果我们的目的在于更好的编程，那么可能需要诸如 Underscore.js 之类的库。或许说打包自己常用的函数功能为一个库，诸如 jQuery

```
function isFunction(value) { return type(value) == "function" }
function isWindow(obj)    { return obj != null && obj == obj.window }
function isDocument(obj)   { return obj != null && obj.nodeType == obj.DOCUMENT_N
function isObject(obj)      { return type(obj) == "object" }
```

我们需要去了解一些故事背后的原因，越来越害怕 GUI 的原因之一，在于不知道背后发生了什么，即使是开源的，我们也无法了解真正的背后发生什么了。对于不是这个

工具、软件的用户来说，开源更多的意义可能在于我们可以添加新的功能，以及免费。如果没有所谓的危机感，以及认为自己一直在学习工具的话，可以试着去打包自己的函数，打包自己的库。

```
var calc={
    add: function(a,b){
        return a+b;
    },
    sub: function(a,b){
        return a-b;
    },
    dif: function(a,b){
        if(a>b){
            return a;
        }else{
            return b;
        }
    }
}
```

然后用诸如 jslint 测试一下代码。

```
$ ./jsl -conf jsl.default.conf
JavaScript Lint 0.3.0 (JavaScript-C 1.5 2004-09-24)
Developed by Matthias Miller (http://www.JavaScriptLint.com)

app.js
/Users/fdhuang/beageek/chapter4/src/app.js(15): lint warning: missing semicolon
    }
........^


0 error(s), 1 warning(s)
```

于是我们需要在第 15 行添加一个分号。

最好的方法还是阅读别人的代码，而所谓的别人指的是一些相对较大的网站的，有

好的开发流程，代码质量也不会太差。而所谓的复杂的代码都是一步步构建上去的，罗马不是一天建成的。

有意思的是多数情况下，我们可能会用原型去开发我们的应用，而这也是我们需要去了解和掌握的地方，

```
function Calc(){

}
Calc.prototype.add=function(a,b){
    return a+b;
};
Calc.prototype.sub=function(a,b){
    return a-b;
};
```

我们似乎在这里展示了更多的 Javascript 的用法，但是这不是一好的关于 Javascript 的介绍，有一天我们还要用诸如 qunit 之类的工具去为我们的 function 写测试，这时就是一个更好的开始。

如果我们乐意的话，我们也可以构建一个类似于 jQuery 的框架，以用来学习。

作为一门编程语言来说，我们学得很普通，在某种意义上来说算不上是一种入门。但是如果我们可以在其他的好书在看到的内容，就没有必要在这里进行复述，目的在于一种学习习惯的养成。

4.3.2 CSS

CSS 有时候很有趣，但是有时候有很多我们没有意识到的用法，这里以 Bootstrap 为例，这是一个不错的 CSS 库。最令人兴奋的是没有闭源的 CSS，没有闭源的 JS，这也就是前端好学习的地方所在了，不过这是一个开源的 CSS 库，虽然是这样叫的，但是称之为 CSS 库显然不合适。

```
a,
a:visited {
  text-decoration: underline;
}
a[href]:after {
  content: " (" attr(href) ")";
```

```
}
abbr[title]:after {
  content: " (" attr(title) ")";
}
a[href^="javascript:"]:after,
a[href^="#"]:after {
  content: "";
}
```

这里有一些有趣的，值得一讲的 CSS 用法。

- 伪类选择器, 如 a:visited 这样需要其他条件来对元素应用样式，用于已访问的链接。
- 属性选择器, 如 a[href] 这样当 a 元素存在 href 这样的属性的时候来寻找应用元素。

其他的还需要去好好了解的就是 **CSS** 的盒模型，作为 CSS 的基石之一。

4.4 CSS 盒模型图

诸如

```
* {
  margin: 0px;
  padding: 0px;
  font-family: Helvetica;
}
```

这样的通用器用来进行全局选择的工具和我们用于抵消某个 body 对于子选择器的影响一样值得注意得多。

4.5 笔记

写博客似乎是一个不错的好习惯，作为一个不是很优秀的写手。对于来说，有时候发现原来能教会别人对于自己的能力来说算是一种肯定。有些时候教会别人才算是自己学会的表现，总会在项目上的时候需要自己去复述工作的一个过程，我们需要整理好我们的思路才能带给别人更多的收获。我们的笔记上总会留下自己的学习的一些过程，有些时候我们想要的只是一点点的鼓励，有时是诸如评论一类，有时可能是诸如访问量。

更多的可能是我们可以重新整理自己的知识，好好复习一下，以便于好好记住，写出来是一个好的过程。

无处不在的三剑客就这样到了这里，写得似乎很多也很少，但是还是没有做出来一个东西，于是我们朝着这样一个方向前进。

5 GNU/Linux

5.1 什么是 Linux

Linux 是一种自由和开放源码的类 UNIX 操作系统内核。目前存在着许多不同的 Linux 发行版，可安装在各种各样的电脑硬件设备，从手机、平板电脑、路由器和影音游戏控制台，到桌上型电脑，大型电脑和超级电脑。Linux 是一个领先的操作系统内核，世界上运算最快的 10 台超级电脑运行的都是基于 Linux 内核的操作系统。

Linux 操作系统也是自由软件和开放源代码发展中最著名的例子。只要遵循 GNU 通用公共许可证, 任何人和机构都可以自由地使用 Linux 的所有底层源代码，也可以自由地修改和再发布。严格来讲，**Linux** 这个词本身只表示 **Linux** 内核，但在实际上人们已经习惯了用 **Linux** 来形容整个基于 **Linux** 内核，并且使用 **GNU** 工程各种工具和数据库的操作系统（也被称为 **GNU/Linux**）。通常情况下，Linux 被打包成供桌上型电脑和服务器使用的 Linux 发行版本。一些流行的主流 Linux 发行版本，包括 Debian（及其衍生版本 Ubuntu），Fedora 和 openSUSE 等。Linux 得名于电脑业余爱好者 Linus Torvalds。

而不是如百度百科所讲的~~Linux 操作系统是 UNIX 操作系统的一种克隆系统。它诞生于 1991 年的 Linux 桌面 [1]10 月 5 日（这是第一次正式向外公布的时间）。以后借助于 Internet 网络，并通过全世界各地计算机爱好者的共同努力，已成为今天世界上使用最多的一种 UNIX 类操作系统，并且使用人数还在迅猛增长。~~

Linux 只是个内核，而不是操作系统，所以在这我们再理解一下操作系统是由什么组成的。

5.2 操作系统

操作系统（英语：Operating System，简称 OS）是管理计算机硬件与软件资源的计算机程序，同时也是计算机系统的内核与基石。操作系统需要处理如管理与配置内存、决定系统资源供需的优先次序、控制输入与输出设备、操作网络与管理文件系统等基本事务。操作系统也提供一个让用户与系统交互的操作界面。操作系统的型态非常多样，

不同机器安装的操作系统可从简单到复杂，可从手机的嵌入式系统到超级计算机的大型操作系统。许多操作系统制造者对它涵盖范畴的定义也不尽一致，例如有些操作系统集成了图形用户界面 (GUI)，而有些仅使用命令行界面 (CLI)，而将 GUI 视为一种非必要的应用程序。

操作系统位于底层硬件与用户之间，是两者沟通的桥梁。用户可以通过操作系统的用户界面，输入命令。操作系统则对命令进行解释，驱动硬件设备，实现用户要求。以现代标准而言，一个标准 PC 的操作系统应该提供以下的功能：

- 进程管理（Processing management）
- 内存管理（Memory management）
- 文件系统（File system）
- 网络通信（Networking）
- 安全机制（Security）
- 用户界面（User interface）
- 驱动程序（Device drivers）

而让我们来看一下两者之间的不同之处，这是一张 linux 的架构图我们可以发现内核只是位于底层。

5.2.1 Linux 架构图

5.2.1.1 用户模式　应用程序（sh、vi、OpenOffice.org等）

复杂库（KDE、glib 等）简单库（opendbm、sin 等）

C 库（open、fopen、socket、exec、calloc 等）

5.2.1.2 内核模式

- 系统中断、调用、错误等软硬件消息
- 内核（驱动程序、进程、网络、内存管理等）
- 硬件（处理器、内存、各种设备）

我们可以发现，由 linux 内核 +shell 可以构成一个操作系统，而 linux 本身只是个内核，也就是图中的内核模式，负责控制系统的这些部分。也就是我们可以发现，Linux 内核构成了一个操作系统除用户界面以外的部分，而 shell 就是这最后的用户界面。

而 linux 内核以外的部分就是由 GNU 计划构成的。

5.2.2 Shell

Shell 是系统的用户界面，提供了用户与内核进行交互操作的一种接口。它接收用户输入的命令并把它送入内核去执行。

实际上 Shell 是一个命令解释器，它解释由用户输入的命令并且把它们送到内核。不仅如此，Shell 有自己的编程语言用于对命令的编辑，它允许用户编写由 shell 命令组成的程序。Shell 编程语言具有普通编程语言的很多特点，比如它也有循环结构和分支控制结构等，用这种编程语言编写的 Shell 程序与其他应用程序具有同样的效果

bash 是一个为 GNU 计划编写的 Unix shell。它的名字是一系列缩写：Bourne-Again SHell --- 这是关于 Bourne shell（sh）的一个双关语（Bourne again / born again）。Bourne shell 是一个早期的重要 shell，由史蒂夫 · 伯恩在 1978 年前后编写，并同 Version 7 Unix 一起发布。bash 则在 1987 年由布莱恩 · 福克斯创造。在 1990 年，Chet Ramey 成为了主要的维护者。

shell 将会是我们在 GNU/linux 中经常用到的经常有到的工具之一，用来操作计算机用的。在迁移到 linux 之前我们可以试用 cygwin 来进行模拟：

Cygwin 是许多自由软件的集合，最初由*Cygnus Solutions*开发，用于各种版本的*Microsoft Windows*上，运行*UNIX* 类系统。*Cygwin*

5.2.3 GCC

GCC（GNU Compiler Collection，GNU 编译器套装），是一套由 GNU 开发的编程语言编译器。它是一套以 GPL 及 LGPL 许可证所发行的自由软件，也是 GNU 计划的关键部分，亦是自由的类 Unix 及苹果电脑 Mac OS X 操作系统的标准编译器。GCC（特别是其中的 C 语言编译器）也常被认为是跨平台编译器的事实标准。

GCC 原名为 GNU C 语言编译器（GNU C Compiler），因为它原本只能处理 C 语言。GCC 很快地扩展，变得可处理 C++。之后也变得可处理 Fortran、Pascal、Objective-C、Java、Ada，以及 Go 与其他语言。

同 shell 一样，对于 GNU/linux 系统而言,GCC 的作用也是无可取代的。当然如果只是一般用途的话，GCC 对于一般用户可能没用，但是在些 GNU/Linux 系统上，我们可能就需要自己编译源码成二进制文件，而没有软件包，因而其重要性是不言而喻的。自然的如果我们自己动手编译 GNU/Linux 操作系统的话，我们会理解其的重要意义。有兴趣的同学可以试一下：Linux From Scratch (LFS)。

5.2.4 启动引导程序

最后，当我们构成以上的那些之后，我们就需要一个引导程序，以便使系统启动，引导进内核。

启动程序（bootloader）于电脑或其他计算机应用上，是指引导操作系统启动的程序。启动程序启动方式与程序视应用机型种类。例如在普通 PC 上，引导程序通常分为两部分：第一阶段引导程序位于主引导记录，用于引导位于某个分区上的第二阶段引导程序，如 NTLDR、GNU GRUB 等。

BIOS 开机完成后，bootloader 就接手初始化硬件设备、创建存储器空间的映射，以便为操作系统内核准备好

正确的软硬件环境。

简单的 bootloader 的虚拟汇编码，如其后的八个指令:

- 0: 将 P 暂存器的值设为 8
- 1: 检查纸带 ({paper tape) 读取器，是否已经可以进行读取
- 2: 如果还不能进行读取, 跳至 1
- 3: 从纸带读取器，读取一 byte 至累加器
- 4: 如为带子结尾，跳至 8
- 5: 将暂存器的值，存储至 P 暂存器中的数值所指定的地址
- 6: 增加 P 暂存器的值
- 7: 跳至 1

但是随着计算机操作系统越来越复杂，位于 MBR 的空间已经放不下引导操作系统的代码，于是就有了第二阶段的引导程序，而 MBR 中代码的功能也从直接引导操作系统变成了引导第二阶段的引导程序。

通常在一个 GNU/Linux 系统中选用 GNUGRUB 做为引导程序，例如 Ubuntu 就是用 GRUB2。

GNU GRUB（简称 ``GRUB''）是一个来自 GNU 项目的启动引导程序。GRUB 是多启动规范的实现，它允许用户可以在计算机内同时拥有多个操作系统，并在计算机启动时选择希望运行的操作系统。GRUB 可用于选择操作系统分区上的不同内核，也可用于向这些内核传递启动参数。

GNU GRUB 的前身为 Grand Unified Bootloader。它主要用于类 Unix 系统；同大多 Linux 发行版一样，GNU 系统也采用 GNU GRUB 作为它的启动器。Solaris 从 10 1/06 版开始在 x86 系统上也采用 GNU GRUB 作为启动器。

以上也就构成了一个简单的操作系统。

5.3 从编译开始

我们以一次编译开始我们的 Linux 学习之旅。

5.3.1 开始之前

- 如果你没有用过 GNU/Linux，我想你需要在虚拟机上安装一个。
- 一个主流的 GNU/Linux 发行版，如 Ubuntu,CentOS,Debian,Mint,OpenSUSE,Fedora 等等。
- 学会如何打开 shell(ps:bash,zsh,sh 等等)。

或者你也可以在 Windows 上安装 Cygwin。

5.3.2 编译 Nginx

1. 下载这个软件的源码包

```
wget http://nginx.org/download/nginx-1.7.4.tar.gz
```

wget 是一个用于下载的软件，当然你也可以用软件，只是用 wget 似乎会比图形界面快哦。

2. 解压软件包

```
tar -vf nginx-1.7.4.tar.gz
```

-vf 的意思是 Extract，也就是解压，而 tar 则是这个解压软件的名字。看上去似乎比 WinRAR 来得复制得多，但是你可以计时一下，从下载到解压完，和你用鼠标比哪个比较快。

3. 到 nginx 目录下

这里需要分两部进行

1). 列出所有文件

```
ls -al
```

```
drwxr-xr-x   15 fdhuang  staff   510B Sep  2 13:44 nginx-1.7.4
-rw-r--r--   1 fdhuang  staff   798K Aug  5 21:55 nginx-1.7.4.tar.gz
```

2). 到 nginx-1.7.4 目录

```
cd nginx-1.7.4
```

4. 配置 nginx

一次简单的配置如下

```
./configure
```

当你熟练以后，你可能和我一样用下面的配置 (注意: 用下面的代码会出错。)

```
./configure --user=www --group=www --add-module=../ngx_pagespeed-1.8.3.4-
beta   -- add- module=../ ngx_cache_purge   -- prefix=/ usr/ local/
nginx --with-pcre --with-http_spdy_module --with-http_ssl_module --
with- http_realip_module   -- with- http_addition_module   -- with-
http_sub_module --with-http_dav_module --with-http_flv_module --
with-http_mp4_module --with-http_gunzip_module --with-http_gzip_static_module
with-http_random_index_module  --with-http_secure_link_module  --
with-http_stub_status_module --with-mail --with-mail_ssl_module --
with-ipv6
```

过程中可能会提示你其中出了多少错误，而这时你便可以很愉快地去用搜索引擎搜索他们。

5.make

这里就会用到 GCC 等等。

```
make
```

6. 运行

如果运行顺利的话，应该可以直接

```
./objs/nginx
```

5.3.3 其他

1. 如果没有 wget,make,gcc 等命令的时候可以用类似于下面的方法安装，

```
sudo apt-get install gcc,make,wget
```

2. 正常情况下一个开源项目都会有一个 README，会告诉你应该如何去做。

6 Arduino

6.1 极客的玩具

Arduino，是一个开放源代码的单芯片微电脑，它使用了 Atmel AVR 单片机，采用了基于开放源代码的软硬件平台，构建于开放源代码 simple I/O 接口板，并且具有使用类似 Java，C 语言的 Processing/Wiring 开发环境。

Arduino 开发板封装了常用的库到开发环境中，可以让用户在开发产品时，将主要注意力放置于所需要实现的功能上，而不是开发的过程中。在为 Arduino 写串口程序时，我们只需要用 Serial.begin(9600) 以 9600 的速率初始化串口，而在往串口发送数据时，可以用 Serial.write(`1') 的方式向串口发送字串'1'。

7 Python

作为一门计算机语言来说，Python 会有下面的特点。

- 语言学习起来容易
- 解决生活中的实际问题
- 支持多学科

我们可以和其他不是脚本语言的语言进行一个简单的对比，如 C，你需要去编译去运行，有时候还需要解决跨平台问题，本来你是在你的 Windows 上运行得好好的，但是有一天你换了一个 Mac 电脑的时候，问题变得很棘手，你甚至不知道怎么去解决问题。我没有用过 MFC，听说很方便，但是在其他平台下就没有一个好的解决方案。这里可能跑得有点远，但是不同的用户可能在不同的平台上，这也就是脚本语言的优势所在了。

7.1 代码与散文

你可能听过，也可能了解过，不过在这里我们可能不会去讲述那些基础的语法的东西，我们想说的是代码格式的重要性，在 html 中你可以这样去写你的代码

```
<html><head><title>This is a Title
</title></head><body><div class="content">
<p>flakjfaklfjalfa</p></div>
</body></html>
```

又或者是 js 的 minify，它可能会使你的代码看起来像是这样的:

```
function NolTracker(b,a){this.pvar=b;this.mergeFeatures(a)}
```

可能的是如果是 python 的话，你可能会遇到下面的问题。。

```
File "steps.py", line 10
   try:
   ^
IndentationError: expected an indented block
```

如果你对 JSLint、Lint 这类的工具有点印象的话，你也可以认为 python 集成了这类工具。整洁的代码至少应该看上去要有整洁的衣服，就好像是我们看到的一个人一样，而后我们才会有一个好的印象。更主要的一点是代码是写给人看的，而衣服更多的时候对于像我这样的人来说，他首先应该是要保暖的，其次对于一个懒的人来说。。。

> 程序应该是具有可读性的短文，它将在计算机上执行，从而解决某些问题

我们需要去读懂别人的代码，别人也需要去读懂我们的代码。计算机可以无条件地执行你那未经编排过的程序，但是人就不是如此了。

```
var calc={add: function(a,b){return a+b;},sub: function(a,b)
{return a-b;},dif: function(a,b){if(a>b){return a;}else{return b;}}}
```

上面的代码相对于下面的代码可读性没有那么多，但是计算机可以无条件地执行上面的代码。上面的代码对于网络传输来说是好的，但是对于人来说并不是如此，我们需要一些工具来辅助我们去读懂上面的代码。如果代码上写得没有一点可读性，诸如函数命名没有多少实际意义，如果我们把前面的函数就成这样:

```
var c={
    a: function(a,b){
        return a+b;
    },
    s: function(a,b){
        return a-b;
    },
    d: function(a,b){
        if(a>b){
            return a;
        }else{
            return b;
        }
    }
}
```

那么只有在我们理解了这个函数是干什么之后才能理解函数是干什么，而不是光看函数名就可以了。

在 Javascript 解决一个函数的办法有很多，在其他一些语言如 Ruby 或者 Perl 中也是如此，解决问题的办法有很多，对于写代码的人来说是一个享受的过程，但是对于维护的人来说并非如此。而这个和 Python 的思想不是很一致的是，Python 设计的理念是

对于特定的问题，只要有一种最好的方法来解决就够了

可读性的代码在今天显得比以前重要的多，以前写程序的时候我们需要去考虑使用汇编或者其他工具来提高程序的效率。

```
        .global _start


        .text
_start:
        # write(1, message, 13)
       mov     $1, %rax              # system call 1 is write
       mov     $1, %rdi             # file handle 1 is stdout
     mov    $message, %rsi        # address of string to output
        mov     $13, %rdx               # number of bytes
```

```
        syscall                 # invoke operating system to do the write

            # exit(0)
          mov     $60, %rax              # system call 60 is exit
           xor     %rdi, %rdi             # we want return code 0
        syscall                 # invoke operating system to exit
    message:
            .ascii  "Hello, world\n"
```

所以上面的代码的可读性在今天新生一代的程序员来说可能没有那么容易理解。芯片运行的速度越来越快，在程序上我们也需要一个越来越快的解决方案，而所谓的越来越快的解决方案指的不是运行速度上，而是开发速度上。如果你没有办法在同样时间内开发出更好的程序，那么你就可能输给你的竞争对手。

7.2 开始之前

我们终于又从一种语言跳到了另外一种语言，我们可能习惯了一种模式，而不敢于去尝试新的东西，这些或许是我们的一些习惯又或者是因为害怕等等。

作为另外一个跨平台能力很强的语言，这里说的是与 Javascript、HTML 比较，或许你会觉得 C 算是最好的，但是我们这里讨论更多的是脚本语言，也就是直接可以运行的。在现在主流的大多数移动平台上，python 也有良好的支持，如 Android,IOS，只是这些算是类 Unix 系统内核，python 还支持之前 Nokia 的 Symbian。

开始之前我们需要确认我们的平台上已经有了 python 环境，也就是可以运行下面的 Hello,World，你可以在网上很多地方看到，最简单的地方还是到官网，又或者是所用移动平台的 store 下载。

7.3 Python 的 Hello,World

Python 的 Hello,World 有两种形式，作为一种脚本语言来说，Javascript 也是一种脚本语言，只是两者之间有太多的不同之处，每个人都会有不同的选择对于一种语言用来作为其的习惯。于是这就是我们的

```
print "Hello,World"
```

当我们把我们的脚本在 shell 环境下运行时

```
>>>  print "Hello,world"
  File "", line 1
    print "Hello,world"
    ^
IndentationError: unexpected indent
>>> print "Hello,world"
Hello,world
>>>
```

如果你没有预料到缩进带来的问题的时候，这就是一个问题了。

和我们在 Javascript 或者是 CSS 里面一样，我们也可以用一个文件的方式来写入我们的代码，文件后缀名是 py，所以创建一个 helloworld.py，输入上面的代码，然后执行

```
python helloworld.py
```

一个理想的结果，或许你试过 C 语言的 helloworld，如果了解过 GCC 的话应该是可以这样的:

```
./a.out
```

也就是执行编译完后的程序，需要注意的是 helloworld.py 没有编译，不过也会输出

```
Hello,world
```

7.4 我们想要的 Hello,World

我们想展示的是如何结合前面学习的内容做一个更有意思的 Hello,World。

```
import cherrypy
class HelloWorld(object):
    def index(self):
        return "Hello World!"
    index.exposed = True

cherrypy.quickstart(HelloWorld())
```

7.5 算法

我们需要去了解算法 (algorithm)，引经据典的话就是这样子:

> a process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer

也就是计算或其他解决问题的操作需要遵循的一个过程或者一套规则，书上还提到的说法是 ------解决问题的诀窍，让我想起了 hack 一词。我们总会去想某些东西是否有一个更快的计算方法，有时候在处理某些问题上也显示了一个好的算法的重要性。

7.6 实用主义哲学

8 Raspberry Pi

8.1 Geek 的盛宴

Raspberry Pi 是一款针对电脑业余爱好者、教师、小学生以及小型企业等用户的迷你电脑，预装 Linux 系统，体积仅信用卡大小，搭载 ARM 架构处理器，运算性能和智能手机相仿。在接口方面，Raspberry Pi 提供了可供键鼠使用的 USB 接口，此外还有千兆以太网接口、SD 卡扩展接口以及 1 个 HDMI 高清视频输出接口，可与显示器或者 TV 相连。

Linux 是一套免费使用和自由传播的类 Unix 操作系统，是一个基于 POSIX 和 UNIX 的多用户、多任务、支持多线程和多 CPU 的操作系统。它能运行主要的 UNIX 工具软件、应用程序和网络协议。它支持 32 位和 64 位硬件。Linux 继承了 Unix 以网络为核心的设计思想，是一个性能稳定的多用户网络操作系统。

Raspberry Pi 相比于一般的 ARM 开发板来说，由于其本身搭载着 Linux 操作系统，可以用诸如 Python、Ruby 或 Bash 来执行脚本，而不是通过编译程序来运行，具有更高的开发效率。

9 HTTP 与 RESTful

9.1 你所没有深入的 HTTP

9.2 REST

REST[ˆrest] 从资源的角度来观察整个网络，分布在各处的资源由 URI 确定，而客户端的应用通过 URI 来获取资源的表征。获得这些表征致使这些应用程序转变了其状态。随着不断获取资源的表征，客户端应用不断地在转变着其状态，所谓表征状态转移。

10 构建基于 CoAP 协议的物联网系统

10.1 CoAP 简介

引自维基百科上的介绍，用的是谷歌翻译。。。

> 受约束的应用协议（COAP）是一种软件协议旨在以非常简单的电子设备，使他们能够在互联网上进行交互式通信中使用。它特别针对小型低功率传感器，开关，阀门和需要被控制或监督远程，通过标准的 Internet 网络类似的组件。COAP 是一个应用层协议，该协议是用于在资源受限的网络连接设备，例如无线传感器网络节点使用。COAP 被设计为容易地转换为 HTTP 与 Web 简化集成，同时也能满足特殊的要求，例如多播支持，非常低的开销，和简单性。多播，低开销，以及简单性是因特网极其重要物联网（IOT）和机器对机器（M2M）设备，这往往是积重难返，有太多的内存和电源，比传统的互联网设备有。因此，效率是非常重要的。COAP 可以在支持 UDP 或 UDP 的模拟大多数设备上运行。

简单地来说，CoAP 简化了 HTTP 协议，只提供了 REST 的四个方法，PUT,GET,POST 和 DELETE，和其与 HTTP 的不同之处在于 ------CoAP 简化了 HTTP 协议。

CoAP 协议中不使用 HTTP 的原因在于: 对于微小的资源受限，在资源受限的通信的 IP 的网络，HTTP 不是一种可行的选择。它占用了太多的资源和太多的带宽。而对于物联网这种嵌入式设备来说，关于资源与带宽，是我们需要优先考虑的内容。

- CoAP 采用了二进制报头，而不是文本报头 (text header)
- CoAP 降低了头的可用选项的数量。
- CoAP 减少了一些 HTTP 的方法
- CoAP 可以支持检测装置

10.2 CoAP 命令行工具

开始之前我们需要安装一个 CoAP 的命令行工具，以方便我们测试我们的代码是否是正确的。

10.2.1 Node CoAP CLI

```
npm install coap-cli -g
```

以便于用这个来测试我们的代码

10.2.2 CoAP 命令行

coap-cli 一共有四个方法

```
Commands:

get                     performs a GET request
put                     performs a PUT request
post                    performs a POST request
delete                  performs a DELETE request
```

我们用coap://vs0.inf.ethz.ch/来作一个简单的测试

```
coap get coap://vs0.inf.ethz.ch/
(2.05)  ************************************************************
I-D
```

也可以用来测试我们现在的最小的物联网系统

```
coap get coap://coap.phodal.com/id/1
(4.06)  [{"id":1,"value":"is id 1","sensors1":19,"sensors2":20}]
```

数据是直接从数据库中读取出来的，然而状态码是错的，忘了刚代码更新。

10.2.3 libcoap

如果我们已经装有 LibCoAP 那么，我们可以直接用自带的两个命令

```
coap-client
coap-server
```

1. 我们简单地起一个 CoAP 服务

```
 coap-server
```

2. 客户端获取数据

```
coap-client -m get coap://localhost
```

返回结果

```
v:1 t:0 tkl:0 c:1 id:37109
This is a test server made with libcoap (see http://libcoap.sf.net)
Copyright (C) 2010--2013 Olaf Bergmann <bergmann@tzi.org>
```

10.3 Hello,World

接着我们便开始试试做一个简单的 CoAP 协议的应用，开始之前我们需要能访问coap://localhost/，于是我们便需要安装一个 Firefox 的插件 Copper。

1. 下载地址: https://addons.mozilla.org/en-US/firefox/addon/copper-270430/
2. 作为测试我们可以访问 coap://vs0.inf.ethz.ch:5683/

10.3.1 Node-CoAP

这是我们这里用到的一个 Node 的扩展 Node-CoAP

> node-coap is a client and server library for CoAP modelled after the http module.

Node-CoAP 是一个客户端和服务端的库用于 CoAP 的模块建模。创建一个 package.json 文件，添加这个库

```
{
    "dependencies":{
```

```
        "coap": "0.7.2"
    }
}
```

接着执行

```
npm install
```

就可以安装好这个库

10.3.2 Node CoAP 示例

接着，创建这样一个 app.js

```
const coap        = require('coap')
    , server  = coap.createServer()

server.on('request', function(req, res) {
  res.end('Hello ' + req.url.split('/')[1] + '\n')
})

server.listen(function() {
  console.log('server started')
})
```

执行

```
node app.js
```

便可以在浏览器上访问了，因为现在什么也没有，所以什么也不会返回。

接着下来再创建一个 client 端的 js，并运行之

```
const coap  = require('coap')
    , req   = coap.request('coap://localhost/World')

req.on('response', function(res) {
  res.pipe(process.stdout)
```

```
})

req.end()
```

就可以在 console 上输出

```
Hello World
```

也就达到了我们的目的，用 CoAP 协议创建一个服务，接着我们应该用它创建更多的东西，如产生 JSON 数据，以及 RESTful。和 HTTP 版的最小物联网系统一样，CoAP 版的最小物联网系统也是要返回 JSON 的。

10.4 数据库查询

10.4.1 Node Module

这说里 Module 的意义是因为我们需要在别的地方引用到 db_helper 这个库，也就是下一小节要的讲的内容。

这样我们就可以在 server.js 类似于这样去引用这个 js 库。

```
var DBHelper = require('./db_helper.js');
DBHelper.initDB();
```

而这样调用的前提是我们需要去声明这样的 module，为了方便地导出函数功能调用。

```
function DBHelper(){
}
DBHelper.initDB = function(){};
module.exports = DBHelper;
```

虽然这里的功能很简单，简单也能做我们想做的事情。

10.4.2 Node-Sqlite3

还是继续用到了 SQLite3，只是这里用到的只是基本的查询和创建。

10.4.2.1 一个简单的 initDB 函数

```
var db = new sqlite3.Database(config["db_name"]);
var create_table = 'create table if not exists basic (' + config["db_table"]

db.serialize(function() {
    db.run(create_table);
    _.each(config["init_table"], function(insert_data) {
        db.run(insert_data);
    });
});
db.close();
```

首先从配置中读取 db_name，接着创建 table，然后调用 underscore 的 each 方法，创建几个数据。配置如下所示

```
config = {
    "db_name": "iot.db",
    "db_table": "id integer primary key, value text, sensors1 float, sensors2
    "init_table":[
        "insert or replace into basic (id,value,sensors1,sensors2) VALUES (1,
        "insert or replace into basic (id,value,sensors1,sensors2) VALUES (2,
    ],
    "query_table":"select * from basic;"
};
```

而之前所提到的 url 查询所做的事情便是

```
DBHelper.urlQueryData = function (url, callback) {
    var db = new sqlite3.Database("iot.db");

    var result = [];
    console.log("SELECT * FROM basic where " + url.split('/')[1] + "=" + url.
    db.all("SELECT * FROM basic where " + url.split('/')[1] + "=" + url.split
        db.close();
        callback(JSON.stringify(rows));
    });
};
```

将 URL 传进来，便解析这个参数，接着再放到数据库中查询，再回调回结果。这样我们就可以构成之前所说的查询功能，而我们所谓的 post 功能似乎也可以用同样的方法加进去。

10.4.3 查询数据

简单地记录一下在 IoT-CoAP 中一次获取数据地过程。

10.4.4 GET

先看看在示例中的 Get.js 的代码，这关乎在后面 server 端的代码。

```
const coap       = require('coap')
     ,requestURI = 'coap://localhost/'
     ,url        = require('url').parse(requestURI + 'id/1/')
     ,req        = coap.request(url)
     ,bl         = require('bl');

req.setHeader("Accept", "application/json");
req.on('response', function(res) {
    res.pipe(bl(function(err, data) {
        var json = JSON.parse(data);
        console.log(json);
    }));

});
req.end();
```

const 定义数据的方法，和我们在其他语言中有点像。只是这的 const 主要是为了程序的健壮型, 减少程序出错，当然这不是 javascript 的用法。

我们构建了一个请求的 URL

```
coap://localhost/id/1/
```

我们对我们的请求添加了一个 Header，内容是 Accept，值是'application/json' 也就是 JSON 格式。接着，便是等待请求回来，再处理返回的内容。

10.4.5 IoT CoAP

10.4.6 判断请求的方法

在这里先把一些无关的代码删除掉，并保证其能工作，so，下面就是简要的逻辑代码。

```
var coap            = require('coap');
var server          = coap.createServer({});
var request_handler = require('./request_handler.js');

server.on('request', function(req, res) {
    switch(req.method){
        case "GET": request_handler.getHandler(req, res);
            break;
    }
});

server.listen(function() {
    console.log('server started');
});
```

创建一个 CoAP 服务，判断 req.method，也就是请求的方法，如果是 GET 的话，就调用 request_handler.getHandler(req, res)。而在 getHandler 里，判断了下请求的 Accept

```
request_helper.getHandler = function(req, res) {
    switch (req.headers['Accept']) {
        case "application/json":
            qh.returnJSON(req, res);
            break;
        case "application/xml":
            qh.returnXML(req, res);
            break;
    }
};
```

如果是 json 刚调用 returnJSON,

10.4.7 Database 与回调

而这里为了处理回调函数刚分为了两部分

```
query_helper.returnJSON = function(req, res) {
    DBHelper.urlQueryData(req.url, function (result) {
        QueryData.returnJSON(result, res);
    });
};
```

而这里只是调用了

```
DBHelper.urlQueryData = function (url, callback) {
    var db = new sqlite3.Database(config["db_name"]);

    console.log("SELECT * FROM basic where " + url.split('/')[1] + "=" + url.
    db.all("SELECT * FROM basic where " + url.split('/')[1] + "=" + url.split
        db.close();
        callback(JSON.stringify(rows));
    });
};
```

这里调用了 node sqlite3 去查询对应 id 的数据，用回调处理了数据无法到外部的问题，而上面的 returnJSON 则只是返回最后的结果，code 以及其他的内容。

```
QueryData.returnJSON = function(result, res) {
    if (result.length == 2) {
        res.code = '4.04';
        res.end(JSON.stringify({
            error: "Not Found"
        }));
    } else {
        res.code = '2.05';
        res.end(result);
    }
};
```

当 resulst 的结果为空时，返回一个 404，因为没有数据。这样我们就构成了整个的链，再一步步返回结果。

在IoT-CoAP中我们使用到了一个 Block2 的东西，于是便整理相关的一些资料，作一个简单的介绍，以及在代码中的使用。

10.5 CoAP Block

CoAP 是一个 RESTful 传输协议用于受限设备的节点和网络。基本的 CoAP 消息是一个不错的选择对于小型载荷如

- 温度传感器
- 灯光开关
- 楼宇自动化设备

然而，有时我们的应用需要传输更大的有效载荷，如 ------更新固件。与 HTTP，TCP 做繁重工作将大型有效载荷分成多个数据包，并确保他们所有到达并以正确的顺序被处理。

CoAP 是同 UDP 与 DLTS 一样是基于数据报传输的，这限制了资源表示 (resource representation) 的最大大小，使得传输不需要太多的分割。虽然 UDP 支持通过 IP 分片传输更大的有效载荷，且仅限于 64KiB，更重要的是，并没有真正很好地约束应用和网络。

而不是依赖于 IP 分片，这种规范基本 COAP 了对 ``块" 选项，用于传输信息从多个资源区块的请求 - 响应对。在许多重要的情况下，阻止使服务器能够真正无状态：服务器可以处理每块分开传输，而无需建立连接以前的数据块传输的其他服务器端内存。

综上所述，块 (Block) 选项提供了传送一个最小的在分块的方式更大的陈述。

10.5.1 CoAP POST

看看在 IoT CoAP 中的 post 示例。

```
const coap     = require('coap')
      ,request  = coap.request
      ,bl       = require('bl')
      ,req = request({hostname: 'localhost',port:5683,pathname: '',method: 'P
```

```
req.setOption('Block2',  [new Buffer('1'),new Buffer("'must'"), new Buffer('2
req.setHeader("Accept", "application/json");
req.on('response', function(res) {
    res.pipe(bl(function(err, data) {
        console.log(data);
        process.exit(0);
    }));

});

req.end();
```

Block2 中一共有四个数据，相应的数据结果应该是

```
{ name: 'Block2', value: <Buffer 31> }
{ name: 'Block2', value: <Buffer 27 6d 75 73 74 27> }
{ name: 'Block2', value: <Buffer 32 33> }
{ name: 'Block2', value: <Buffer 31 32> }
```

这是没有解析的 Block2，简单地可以用

```
_.values(e).toString()
```

将结果转换为

```
Block2,1
Block2,'must'
Block2,23
Block2,12
```

接着按 ``," 分开，

```
_.values(e).toString().split(',')[1]
```

就有

```
[ '1', '\'must\'', '23', '12' ]
```

便可以很愉快地将其 post 到数据库中了，

10.5.2 JSON 请求

在做 IoT-CoAP 的过程中只支持 JSON，查阅 CoAP 的草稿时发现支持了诸多的 Content Types。

10.5.3 CoAP Content Types

以下文字来自谷歌翻译:

> 互联网媒体类型是通过 HTTP 字符串标识，如 ``application/xml''。该字符串是由一个顶层的类型 ``applicaion'' 和子类型的 ``XML''。为了尽量减少使用这些类型的媒体类型来表示的开销消息有效载荷，COAP 定义一个标识符编码方案互联网媒体类型的子集。预计这桌将可扩展标识符的值的 IANA 维护。内容类型选项被格式化为一个 8 位无符号整数。初始映射到一个合适的互联网媒体类型标识符表所示。复合型高层次类型（multipart 和不支持消息）。标识符值是从 201-255 保留的特定于供应商的，应用程序特定的或实验使用和不由 IANA。

下面是 HTTP 的标识符及类型

Internet media type Identifier	
text/plain (UTF-8)	text/xml (UTF-8)
0	1

而在 CoAP 中只有简单地几个

Media type

Encoding

Id.

Reference

text/plain;

-

0

[RFC2046][RFC3676][RFC5147]

charset=utf-8

application/

-

40

[RFC6690]

link-format

application/xml

-

41

[RFC3023]

application/

-

42

[RFC2045][RFC2046]

octet-stream

application/exi

-

47

[EXIMIME]

application/json

-

50

[RFC4627]

简单地说就是:

诸如 `application/json` 的 `Content Types` 在 `CoAP` 中应该是 `50`。如上表所示的结果是其对应的结果，这样的话可以减少传递的信息量。

10.6 CoAP JSON

于是在一开始的时候首先支持的便是 ``application/json'' 这样的类型。

首先判断请求的 header

```
request_helper.getHandler = function(req, res) {
    switch (req.headers['Accept']) {
        case "application/json":
            qh.returnJSON(req, res);
            break;
        case "application/xml":
            qh.returnXML(req, res);
            break;
    }
};
```

再转至相应的函数处理，而判断的依据则是 Accept 是不是 ``application/json''。

```
registerFormat('text/plain', 0)
registerFormat('application/link-format', 40)
registerFormat('application/xml', 41)
registerFormat('application/octet-stream', 42)
registerFormat('application/exi', 47)
registerFormat('application/json', 50)
```

对应地我们需要在一发出请求的时候设置好 Accept，要不就没有办法返回我们需要的结果。

```
req.setHeader("Accept", "application/json");
```

10.6.1 返回 JSON

在给 IoT CoAP 添加了 JSON 支持之后，变得非常有意思，至少我们可以获得我们想要的结果。在上一篇中我们介绍了一些常用的工具 ------CoAP 命令行工具集。

10.6.2 CoAP 客户端代码

开始之前我们需要有一个客户端代码，以便我们的服务端可以返回正确的数据并解析

```
var coap = require('coap');
var requestURI = 'coap://localhost/';
var url = require('url').parse(requestURI + 'id/1/');
console.log("Request URL: " + url.href);
var req = coap.request(url);
var bl = require('bl');

req.setHeader("Accept", "application/json");
req.on('response', function(res) {
    res.pipe(bl(function(err, data) {
        var json = JSON.parse(data);
        console.log(json);
    }));

});

req.end();
```

代码有点长内容也有点多，但是核心是这句话：

```
req.setHeader("Accept", "application/json");
```

这样的话，我们只需要在我们的服务端一判断，

```
if(req.headers['Accept'] == 'application/json') {
     //do something
 };
```

这样就可以返回数据了

10.6.3 CoAP Server 端代码

Server 端的代码比较简单，判断一下

```
if (req.headers['Accept'] == 'application/json') {
        parse_url(req.url, function(result){
            res.end(result);
        });
        res.code = '2.05';
    }
```

请求的是否是 JSON 格式，再返回一个 205，也就是 Content，只是这时设计是请求一个 URL 返回对应的数据。如

```
coap://localhost/id/1/
```

这时应该请求的是 ID 为 1 的数据，即

```
[ { id: 1, value: 'is id 1', sensors1: 19, sensors2: 20 }]
```

而 parse_url 只是从数据库从读取相应的数据。

```
function parse_url(url ,callback) {
    var db = new sqlite3.Database(config["db_name"]);

    var result = [];
    db.all("SELECT * FROM basic;", function(err, rows) {
        callback(JSON.stringify(rows));
    })
}
```

并且全部都显示出来，设计得真是有点不行，不过现在已经差不多了。