

IoT・新技術応用研究会 ミニセミナー
「Node.jsとRaspberry Piを用いた
IoTシステムの構築実習」

名古屋市工業研究所
プロジェクト推進室 斎藤 直希
saito.naoki@nmiri.city.nagoya.jp

この資料について

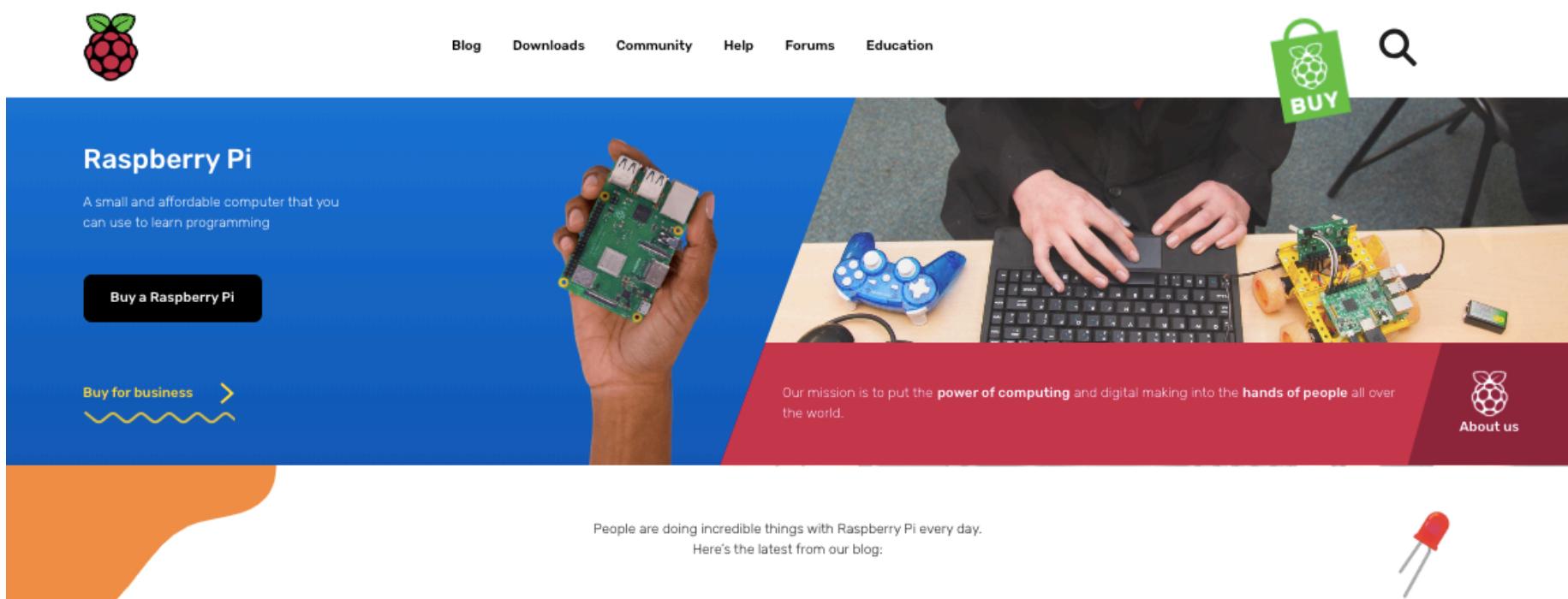
- 本資料を許可なく複製・改変・再配布することを禁じます.
- Raspberry Pi は英国 Raspberry Pi 財団の商標です.
- Node.jsは、Joyent, Inc.の商標です.
- その他、本資料中のすべてのブランド名および製品名は個々の所有者の登録商標もしくは商標です.
- 本資料の内容は予告なく改定することがあります.

予定

- オリエンテーション
 - Node.js について
 - 実習の内容など
- 実習(～16:30)
 - 電子デバイスの制御
 - スイッチ, カメラ, 温度センサ(I2C)
 - クラウドサービスとの連携
 - 簡易HTTPサーバの構築
- まとめ(16:30～17:00)

目標

- ・「ラズベリーパイ」でNode.js(JavaScript)を使い、各自のIoTシステムを考え、実現する



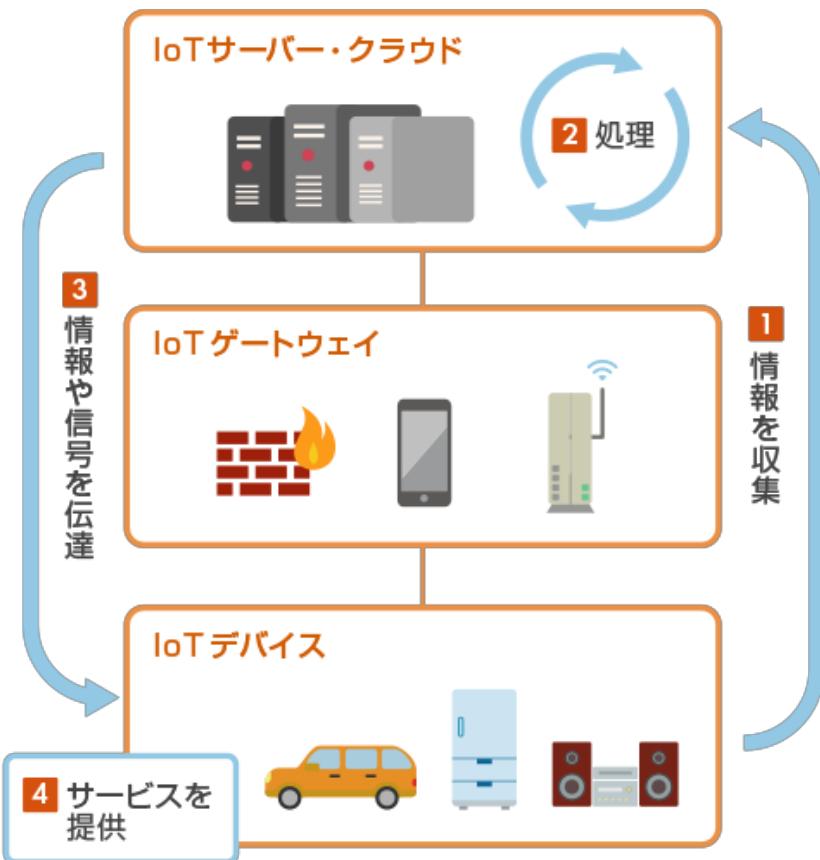
Raspberry Pi — Teach, Learn, and Make with Raspberry Pi
<https://www.raspberrypi.org/>

ラズベリーパイ(Raspberry Pi)

- 計算機科学の学習促進を目的として、英國ラズベリーパイ財団によって開発された小型コンピュータ
 - Linuxなど、パソコンと同様のソフトウェア環境が動作する
 - <https://www.raspberrypi.org/>
- 情報源
 - <https://www.raspberrypi.org/help/>



IoTシステムの構成要素



1. サーバ
– データの処理や蓄積
 - クラウドやLAN内など
2. ネットワーク
– 中継機器、データの転送
 - ルータ、スイッチなど
3. デバイス
– データの取り込みや機器の制御
 - スイッチ、センサ、LED、モータなど

IoTデバイスの構造

<http://www.intellilink.co.jp/article/column/IoT-pp01.html>

JavaScript

- プログラミング言語の一種
 - Javaと名前が似ているが、別の言語
 - ECMAScript-262 として標準化
 - ISO/IEC 16262, JIS X 3060 にもなっている
<https://www.ecma-international.org/publications/standards/Ecma-262.htm>
- 用途
 - ウェブブラウザ上で動作し、動的なウェブサイトや「リッチインターネットアプリケーション」の実現
 - (主にウェブ)サーバ上のJavaScript実行環境上で動作し、各種のサーバ処理の実現

JavaScript の経緯

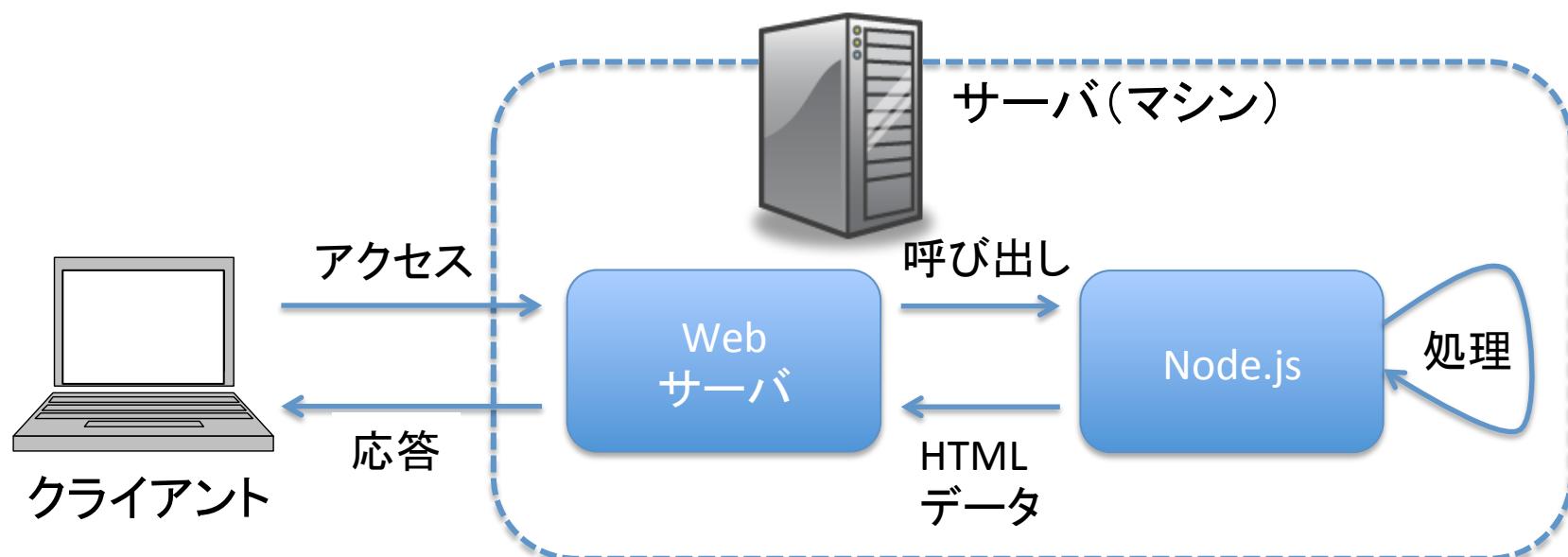
- 1995年, ネットスケープコミュニケーションズ社のブレンダン・アイクにより開発
 - 当初はNetscape Navigator(今の Mozilla)で動作
- 2005年, 非同期通信技術Ajaxにより高機能なウェブアプリケーション開発言語として注目
 - Gmail, Google Map
- 2009年, ライアン・ダールによりNode.jsが開発され, Webサーバ側のネットワークプログラム開発用の言語としても利用



Node.js

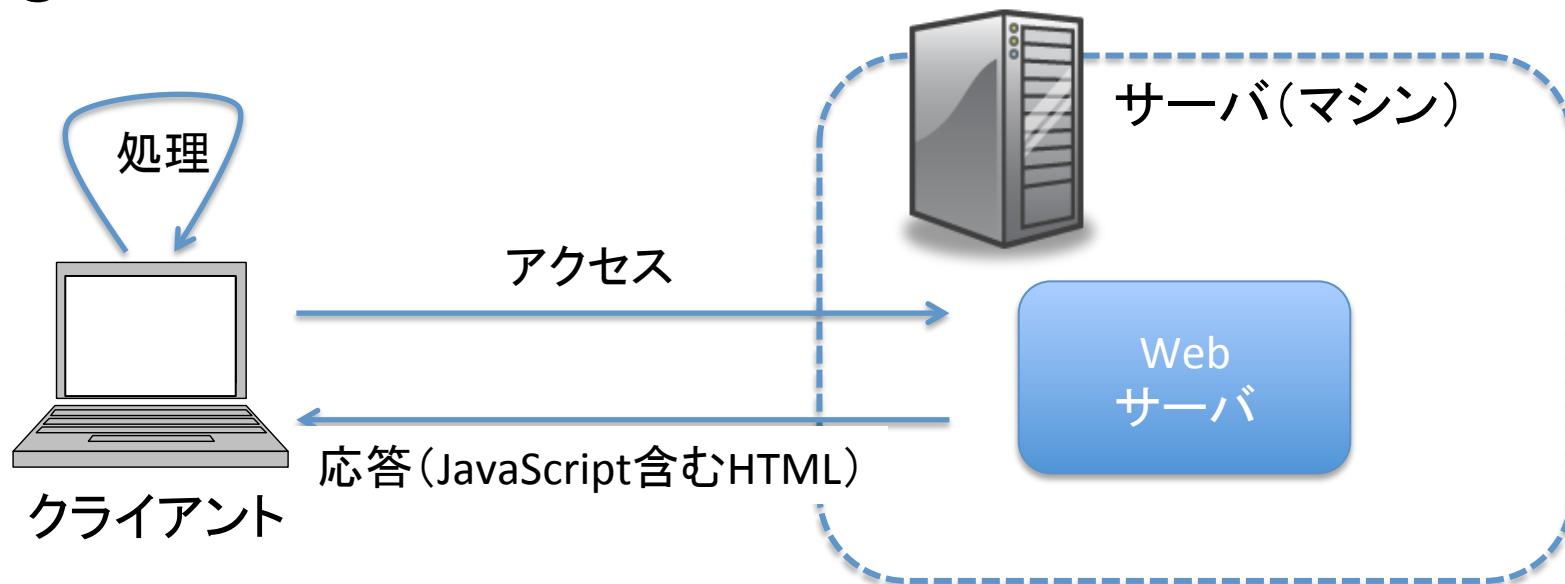
(出典) <https://nodejs.org/en/about/resources/>

- ・ プログラミング言語JavaScriptの動作環境
 - 主にWebサーバのプログラムを実現する目的で開発された(サーバ・サイドJavaScript)
 - JavaScriptプログラム実行環境としても利用可能



(比較) クライアント・サイドJavaScript

- JavaScriptのプログラムがクライアント(ウェブブラウザ)で実行される
- JavaScriptプログラムの目的: HTMLやCSSを書き換える



特徴

- ノンブロッキングのイベント駆動
 - ノンブロッキング : 終わるまで待たない
 - イベント駆動 : プログラム(コールバック)がイベント発生時に呼び出される
 - イベントがたくさん発生したら、たくさん呼び出される(はず)
 - どれだけたくさん実行できるかが実行環境の良し悪しとなる
 - C10k問題(クライアント1万台問題)
 - Node.jsはたくさんのイベントを処理できる(ことを売りにしている)
- シングルスレッドで動作
 - イベントループ(イベントを監視して必要な処理を起動する繰り返し処理)が内部で動作
 - プロセスを生成してマルチコアの性能を活かす設計も一応可能

Node.js の向き・不向き

- 特徴が活きる用途
 - 処理が短時間でイベント処理が重要なアプリ
 - 大量アクセスのあるリアルタイムなネットワークのプログラミング
 - 例) チャットアプリ
 - シングルCPUのサーバー
 - シングルプロセス・シングルスレッドで動作するため、比較的性能の小さいサーバ上でもパフォーマンスを発揮できる
- 不向きな用途
 - CPU負荷の高い処理
 - イベントループが回らない状態になり、イベントが処理できない
 - マルチコア
 - そのままではハードウェア性能を十分使い切ることができない
 - クラスター・モジュールを使ってマルチプロセスで動かせるが、マルチプロセスを意識したプログラミング設計が必要
 - 多数のイベント処理が必要ないサーバ
 - 画像などの静的コンテンツを提供するサーバなど

Node.jsで作られているサービス

- Paypal(決済サービス)
 - サーバー側の開発言語をJavaからNode.jsに切り替えることでフロント側開発との隔たりが減った
- Uber(タクシーの配車サービス)
 - 基幹システムにNode.jsを使用
 - 世界中で使用されているサービスには膨大なネットワークシステムの運用が必要で、Node.jsが採用された
- LinkedIn(ビジネス特化型SNS)
 - モバイル版にNode.jsが使用されている
 - 以前はRuby on Railsを使用していたが、アクセス速度を上げるためにNode.jsに移行

本日の実習予定

1. ハードウェアの制御

- LEDの点灯・消灯
- 温度センサの取り込み
- カメラからの画像取り込み

2. ネットワークサービスの利用

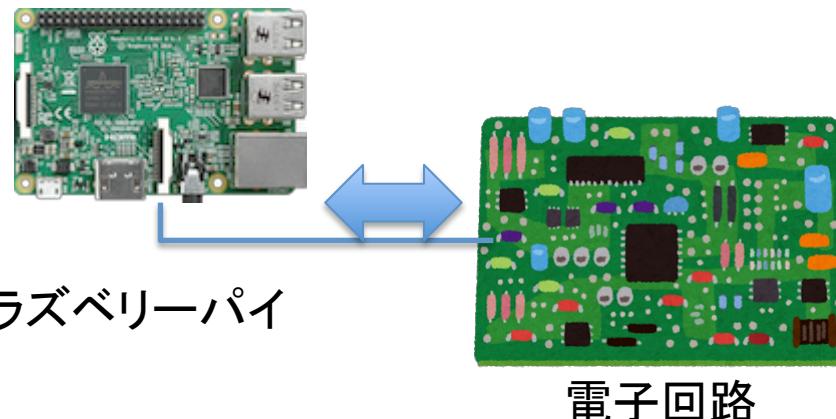
- M2Xへの温度データ送信
- IFTTT(イフト)を利用したネットワークサービスとの連携

3. ウェブサーバ機能の実現

✓ 興味・関心のあるところを取り組む

実習の手順

1. 電子回路を用意する(つくる)
 2. ラズベリーパイと回路とをつなぐ
 3. テキストエディターなどでプログラムをつくる
 4. ターミナルアプリなどでプログラムを動かす
- 最初に1, 2を実施する。それから 3, 4の繰り返し



```
const http = require('http');

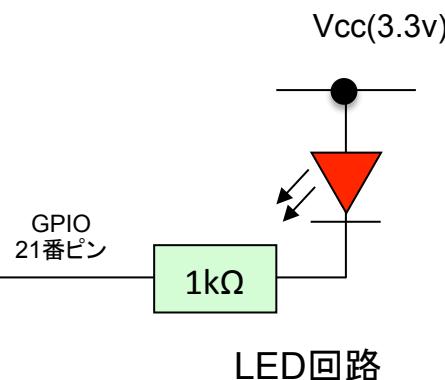
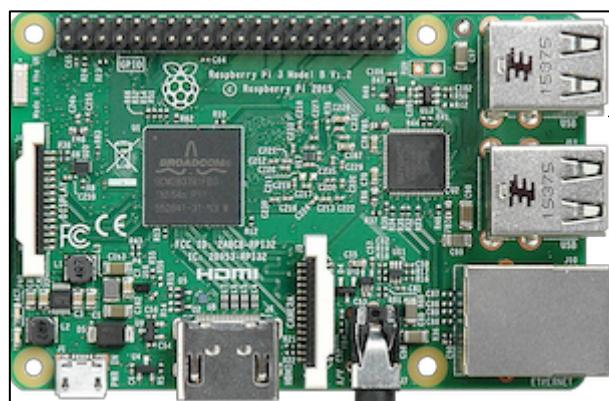
http.createServer(function (request, response) {
  response.writeHead(200, {'Content-Type': 'text/plain'});
  response.end('Hello World\n');
}).listen(3000);

console.log('Server running at http://127.0.0.1:3000/');
```

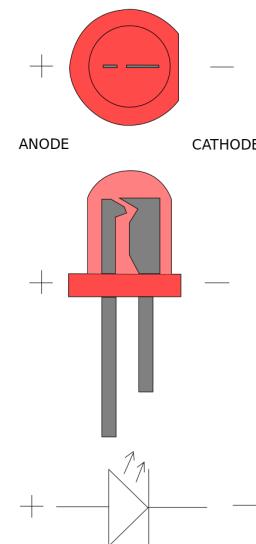
プログラムファイル

LED回路の作製

- まずはLEDを点灯・消灯する回路を作ってみる
 - LEDは向きに注意する



出力ポートは
LowでLEDが点灯



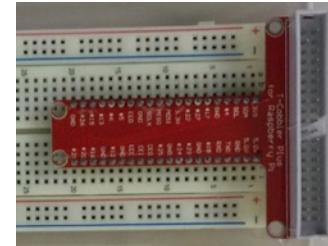
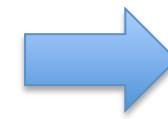
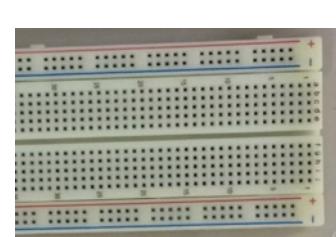
ラズベリーパイ

(画像引用)
[https://en.m.wikipedia.org/wiki/
File:Raspberry_Pi_3_Model_B.png](https://en.m.wikipedia.org/wiki/File:Raspberry_Pi_3_Model_B.png)

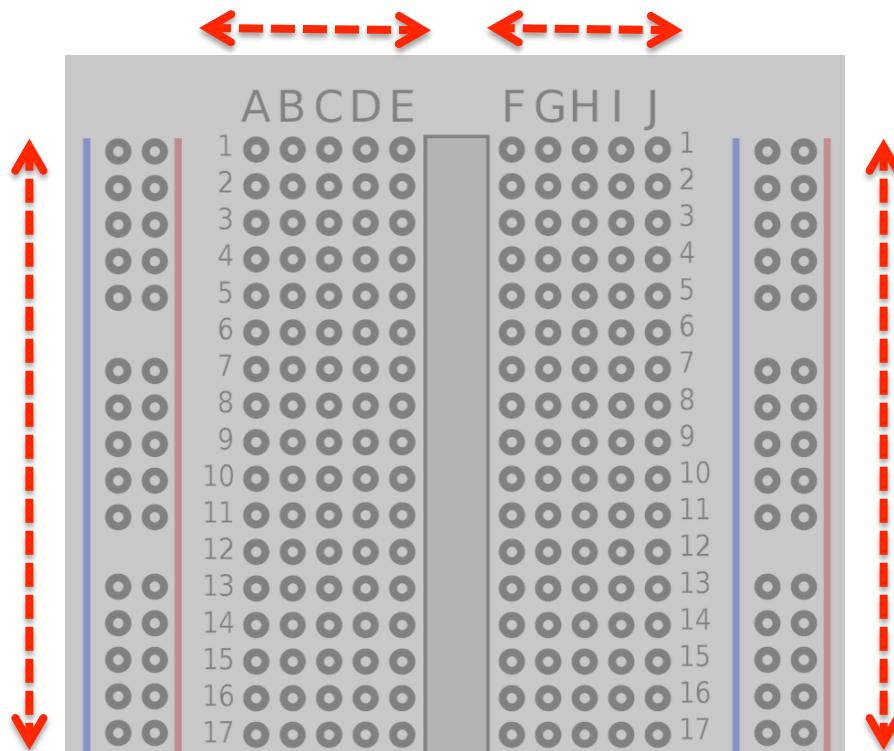
(画像引用) [https://commons.wikimedia.org/
wiki/File:%2B_of_Led.svg](https://commons.wikimedia.org/wiki/File:%2B_of_Led.svg)

ブレッドボードの接続

- ラズベリーパイの電源を入れる前に、ブレッドボードを接続



ブレッドボードの見方



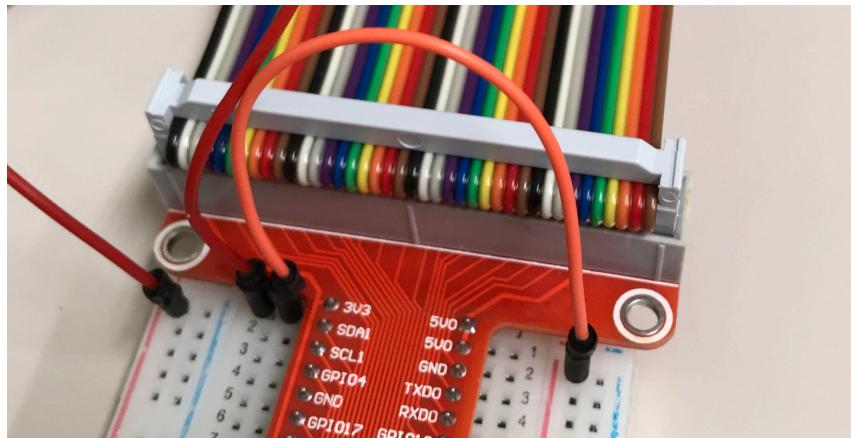
- 中央部分は横方向に導通
 - 例えば1番について
 - A,B～,Eが導通
 - F～Jが導通
 - A～EとF～Jは導通しない
 - 2番以降も同様
- 両端は縦方向に導通
 - 上端から下端まで導通
 - 電源(赤)とGND(青)を接続することが多い

(引用) <https://upload.wikimedia.org/wikipedia/commons/thumb/d/df/Breadboard.svg/2000px-Breadboard.svg.png>

電源(3.3V), GNDの接続

電源の接続

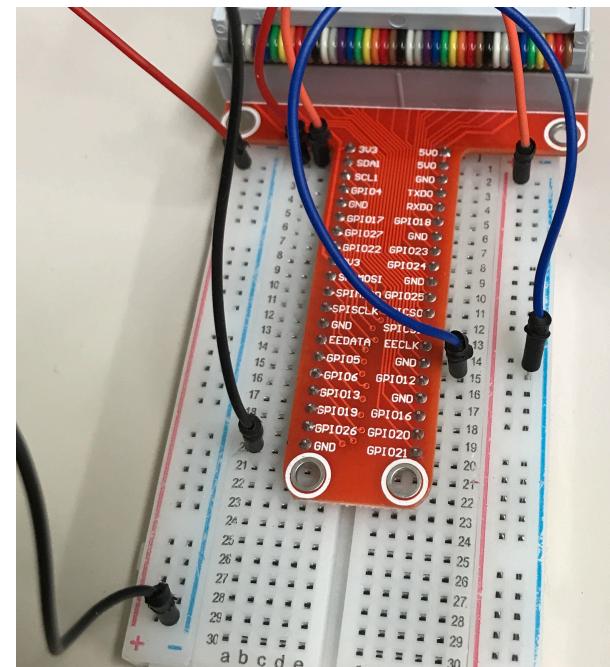
- 「3V3」の記載された端子を
ブレッドボード両端の電源
ライン(赤)に接続



電源とGNDは短絡させないこと

GNDの接続

- 「GND」の記載のある端子
をブレッドボード両端の
GNDライン(青)に接続

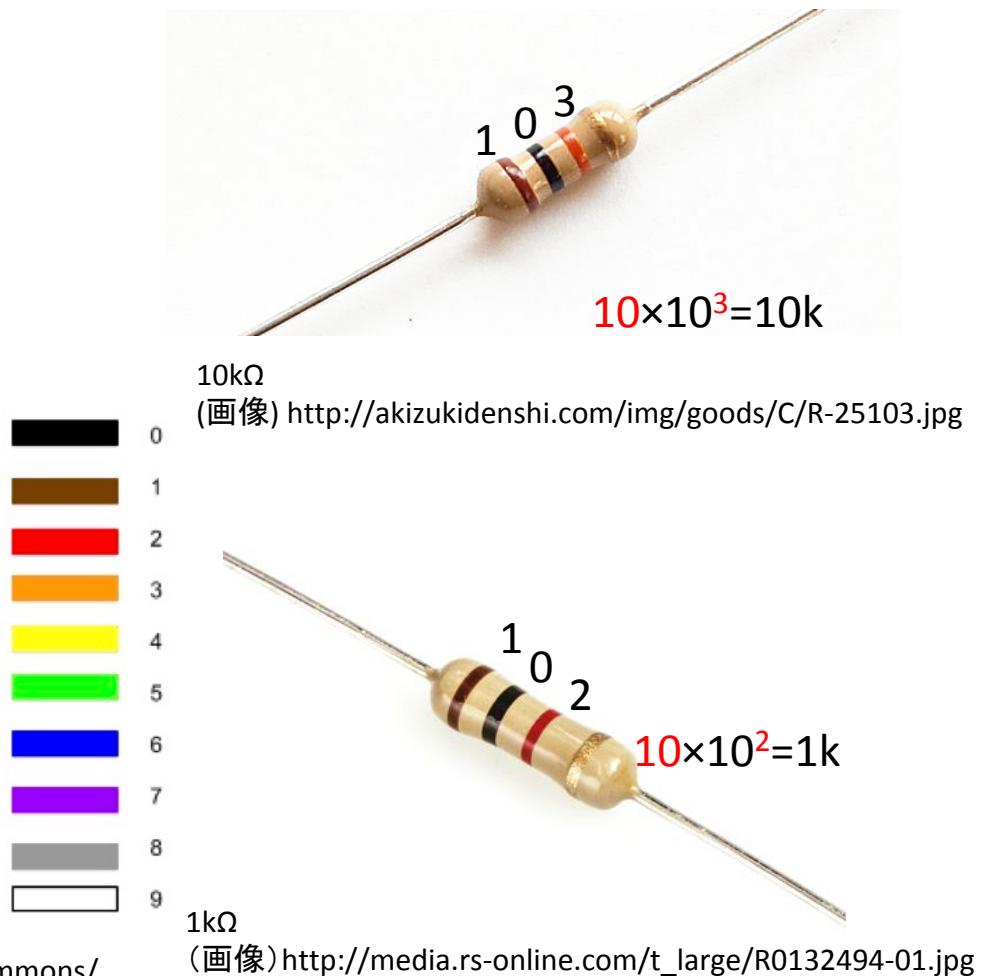


抵抗の見方

- 4本の色分けされた線で抵抗の値(電流の流れにくさをあらわす値)を表現する
 - そのうち3本の色によって有効数字を示す
 - 残り1本は許容差を示す
 - 金は $\pm 5\%$ を表す

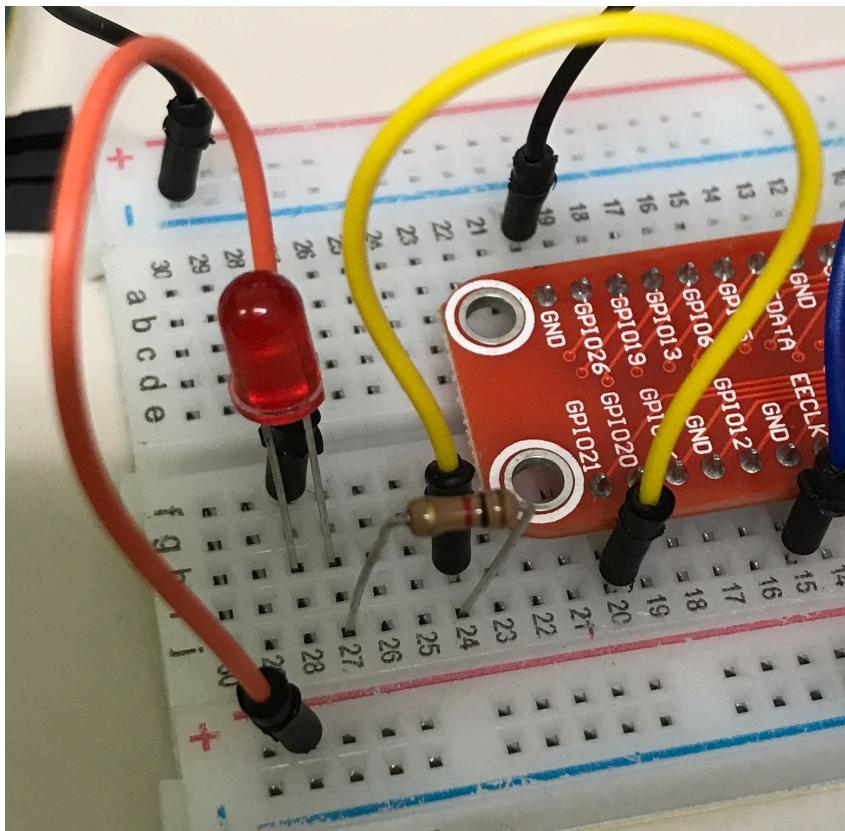
実際にはモノにより5本の線で表現したりなどいろいろなバリエーションがある

<https://upload.wikimedia.org/wikipedia/commons/e/ea/ResistorColorcode.jpg>



LEDの回路

接続のようす

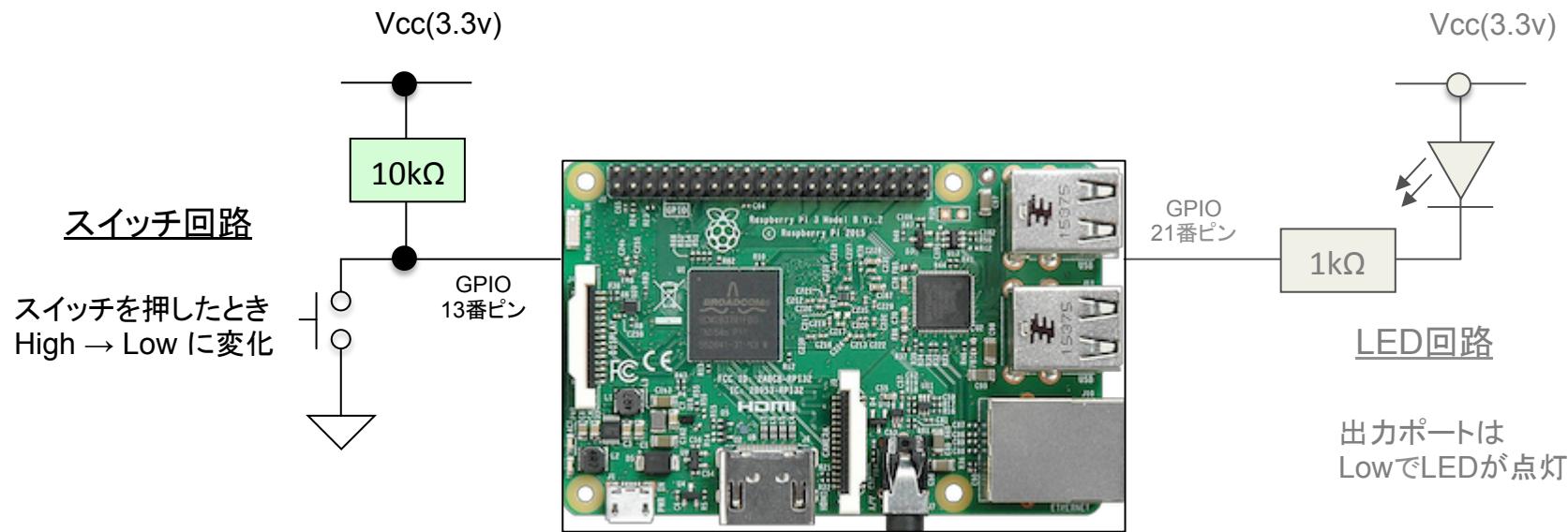


接続の注意

- 「GPIO21」と記載のある端子に接続する
- 「電源>LED>抵抗(1kΩ)>GPIO21 の順に接続」
 - LEDと抵抗は逆順でも良い
- LEDの向きに注意
- 電源とGNDを間違えない

スイッチ回路の作製

- 次に、LEDの回路はそのままで、スイッチの状態を読み取る回路を追加する



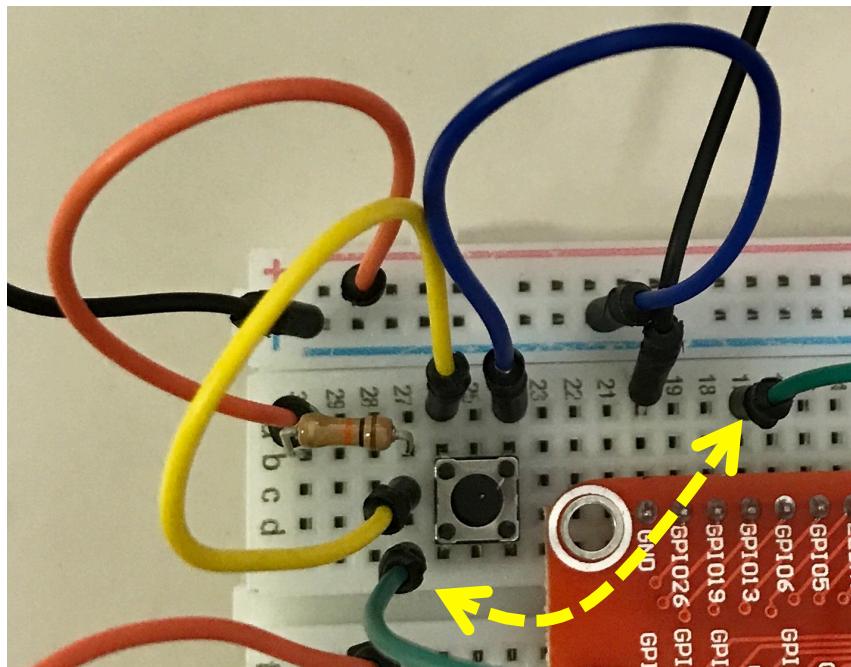
ラズベリーパイ

(画像引用)
[https://en.m.wikipedia.org/wiki/
File:Raspberry_Pi_3_Model_B.png](https://en.m.wikipedia.org/wiki/File:Raspberry_Pi_3_Model_B.png)

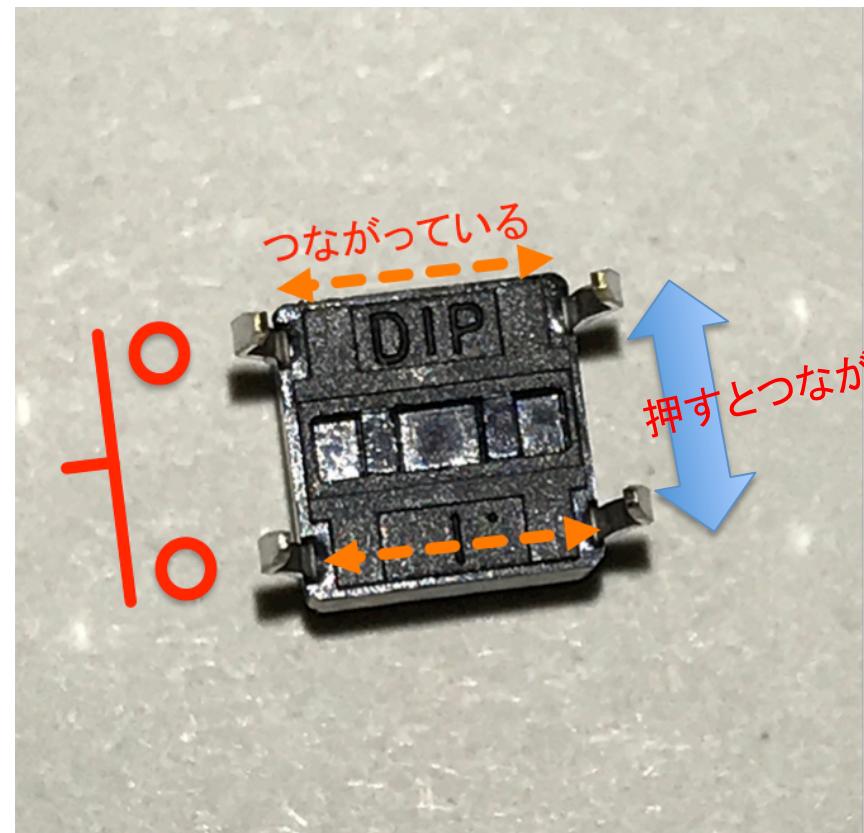
スイッチの回路

接続のようす

- 接続注意

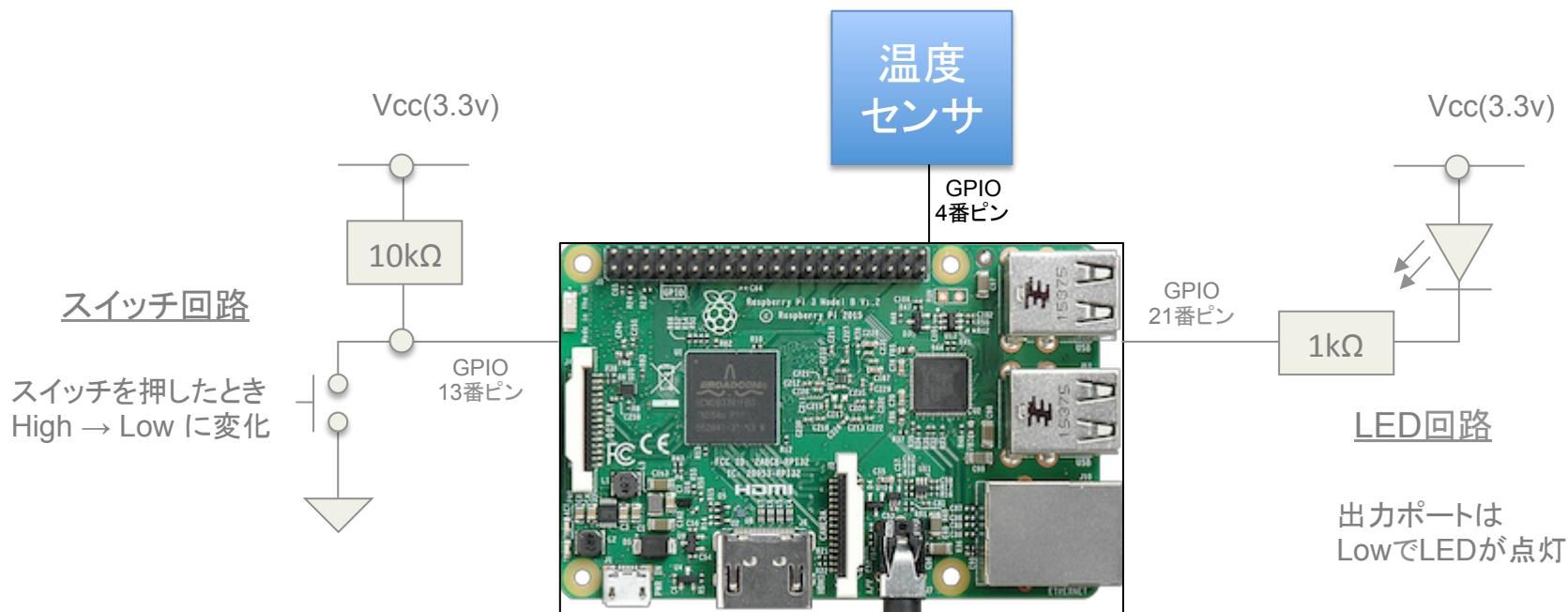


スイッチの見方



温度センサの接続

- 温度を測定する回路を追加する.
- 温度センサとしてDHT11モジュールを使用する
(データシート) <https://akizukidenshi.com/download/ds/aoSong/DHT11.pdf>



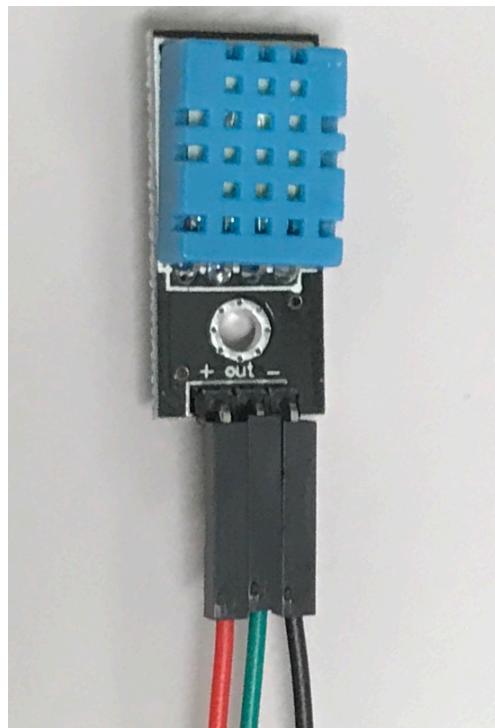
ラズベリーパイ

(画像引用)
[https://en.m.wikipedia.org/wiki/
File:Raspberry_Pi_3_Model_B.png](https://en.m.wikipedia.org/wiki/File:Raspberry_Pi_3_Model_B.png)

温度センサの回路

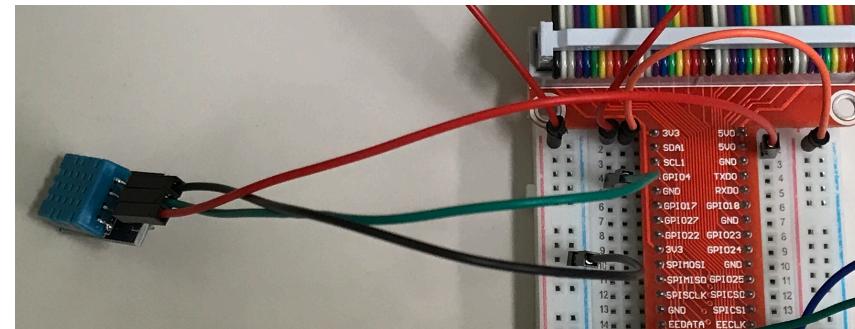
ケーブルを接続

- 「+」: Vcc(5v) ※電圧注意
- 「-」: GND
- 「out」: センサ出力

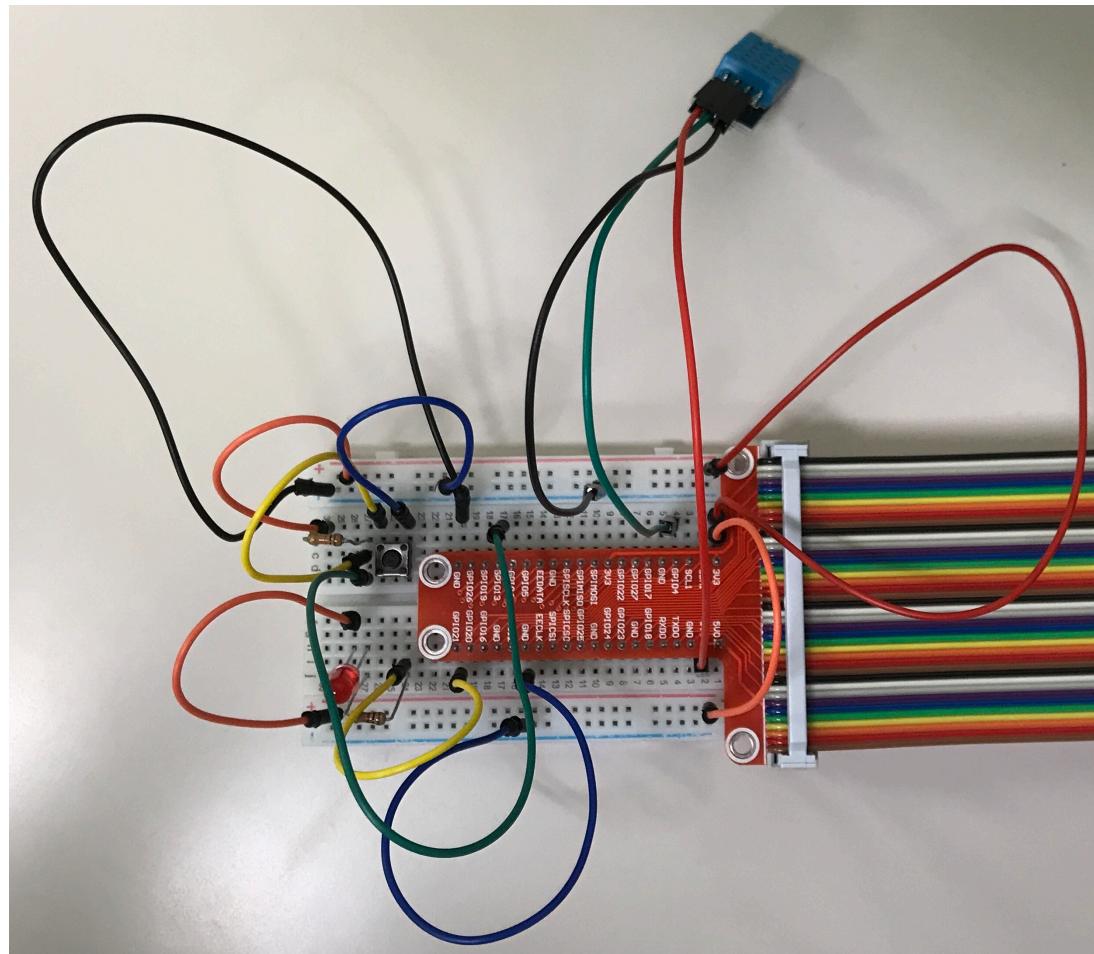


ブレッドボードに接続

- 電源は「5v0」の記載の端子に接続する
- センサの出力は「GPIO4」の端子に接続する

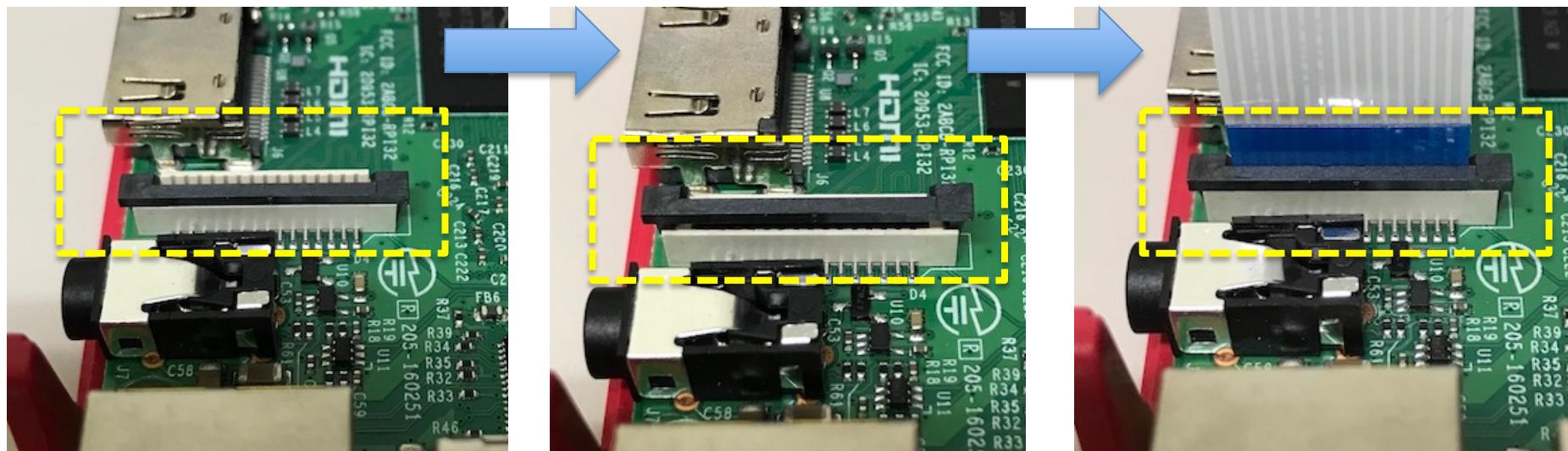


全体の接続のようす



カメラモジュールの接続

- ・ コネクタの爪を引き上げてからケーブルを挿入する
 - 向きに注意



ケーブル等をセット後、電源ON



※ 間違えてパソコンの電源ケーブルを抜かないように

回路の動作確認

- ターミナルアプリで確認
 - LEDの点灯確認
 - スイッチの入力確認
 - 温度センサの動作確認
- 確認できない場合
 - 接続が正しいかどうか
 - 接続がゆるくないか
 - スイッチやLEDの取り付け向きが合っているか

```
# 21番ピンを出力ピンに設定する
$ gpio -g mode 21 out
$ gpio -g mode 13 in

# ピンのレベルをLowにしてLEDを点灯する。
# 消灯は0を1にする。
$ gpio -g write 21 0

# スイッチの入力
# (スイッチを離した状態で)
$ gpio -g read 13
1

# スイッチの入力
# (スイッチを押した状態で)
$ gpio -g read 13
0

# LED の消灯
$ gpio -g write 21 1

# 温度センサの動作確認
$ node ~/iot_training_20181204/dht11_test.js
temp: 24.0° C, humidity: 28.0%
```

カメラの動作確認

- ターミナルアプリで確認
 - 画像の撮影, 表示
 - 動画の撮影, 表示
- 確認できない場合
 - ケーブルの向きがあるて
いるかどうか
 - ケーブルがコネクタに奥
まで差し込まれているか

```
# カメラ画像を撮影  
$ raspistill -o sample.jpg
```

```
# 画像を表示  
$ gpicview sample.jpg
```

```
# 動画を撮影  
$ raspivid -o video.h264 -t 5000
```

```
# 動画の再生  
$ omxplayer video.h264
```

プログラムの作成・実行

1. テキストエディタなどを用いて、右のように入力し、適当な名称でファイルを保存する
 - ここでは hello.js とする
2. ターミナルアプリで、保存したファイルをパラメータとして node コマンドを実行する
3. 結果が表示される

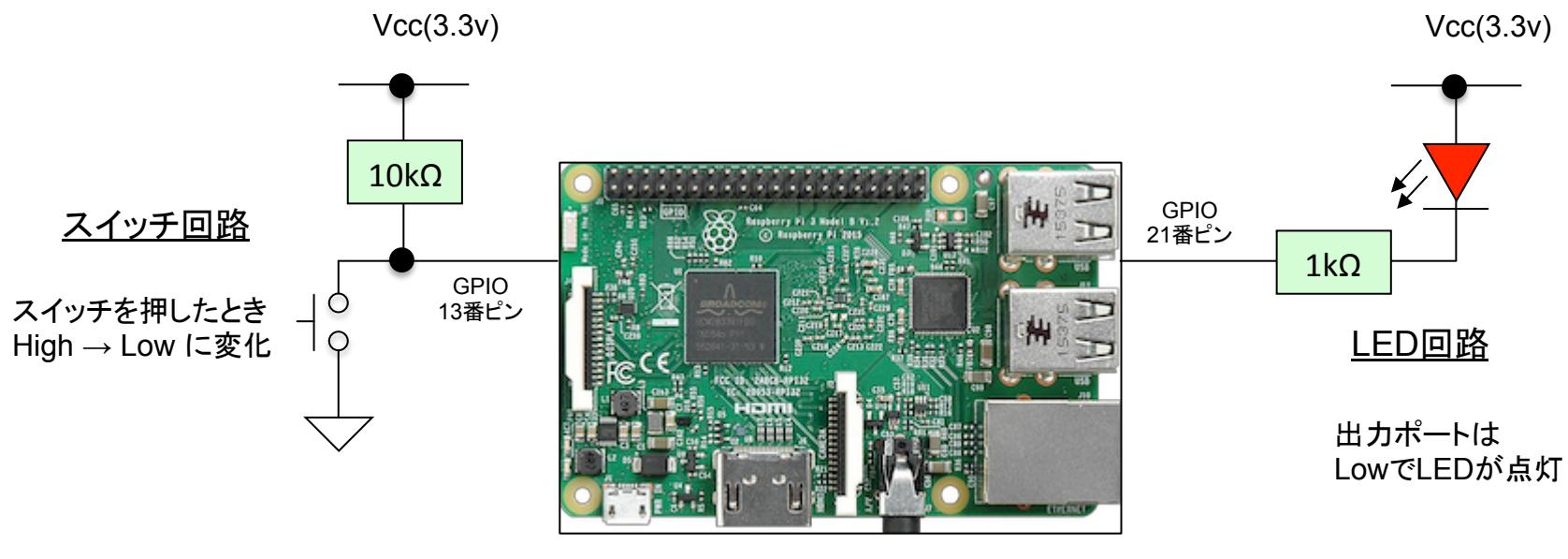
```
// コンソールにメッセージを出力する  
console.log('Hello World!');
```

hello.js の実行

```
$ node hello.js  
Hello World!
```

実習1-1: LEDチカラチカラ

- LEDの点灯および消灯を行う
 - スイッチの状態を見てLEDを点灯・消灯する
 - 時間の経過を見てLEDを点灯・消灯する



ラズベリーパイ

(画像引用)
[https://en.m.wikipedia.org/wiki/
File:Raspberry_Pi_3_Model_B.png](https://en.m.wikipedia.org/wiki/File:Raspberry_Pi_3_Model_B.png)

onoffモジュールを利用した LEDの点灯, 消灯

- onoff モジュール
 - GPIOおよび割り込み処理のためのモジュール
<https://www.npmjs.com/package/onoff>
- キーワード
 - var
 - 変数を宣言する
 - require
 - 他のモジュールを呼び出す
 - new
 - 新しいインスタンスを生成する
 - writeSync
 - onoff の機能で, ピンの出力を設定する
 - ここでは GPIO 21ピンを Low にして LED を点灯
 - 消灯する場合は0を1にする

```
// LEDを点灯する(led_onoff1.js)
//
// 書き方についての補足 :
// 行頭のスラッシュ2本はコメント
// 行末のセミコロン ; は文の区切り

// onoff モジュールの Gpio を取得
var gpio = require('onoff').Gpio;

// GPIO 21ピンを出力ピンに設定
var led = new gpio(21, 'out');

// GPIO 21ピンをLow出力にする(つまり点灯)
led.writeSync(0);
```

```
# 実行手順
$ node led_onoff.js
```

onoffモジュールを利用した ボタンスイッチからの読み出し

- ボタンの宣言を追加
 - 第2引数で'in'を指定して入力であることを示す
- キーワード
 - readSync
 - GPIOピンの状態読み出し
 - 0または1
 - while
 - 条件が成立したときに繰り返す時に使う
 - if
 - 条件成立時に1回だけ何かを行う時に使う
 - ここはピンの入力に応じて出力を変えるのに使用

```
// LEDの点灯・消灯（ボタンの状態で変わる）
// (led_onoff2.js)

var gpio = require('onoff').Gpio;
var led = new gpio(21, 'out');

// GPIO 13ピン（ボタンスイッチ用）を宣言
var buttonSw = new gpio(13, 'in');

while(1) { // ずっと続ける（よくない書き方）
    // スイッチが押されていたらLEDを点灯
    // そうでなければ消灯する
    if(buttonSw.readSync() == 0) {
        led.writeSync(0);
    }
    else {
        led.writeSync(1);
    }
}
```

```
# 実行（終了するときは Ctrl-Cを押す）
$ node led_onoff2.js
```

ボタンの状態変化でLEDトグル動作

```
// ボタンの状態変化でLEDの点灯・消灯を行う
// (led_onoff3.js)

var gpio = require('onoff').Gpio;
var led = new gpio(21, 'out');

// GPIO 13ピンの宣言
// 立ち下がりの変化を検出することを示す
var buttonSw = new gpio(13, 'in', 'falling');

// ハードウェア割り込みを監視する
// 同時に検出した時の処理を指定
buttonSw.watch(function (err, value) {
  if (err) { // エラーの場合の処理
    console.error('error:', err);
    return;
  }
  // ボタンの状態をコンソール出力
  console.log('button pushed:', value);
  // LEDの出力を反転
  if (led.readSync() === 0) {
    led.writeSync(1);
  } else {
    led.writeSync(0);
  }
});
```

```
}());

// Ctrl-Cで終了させた時の後始末処理
function unexportOnClose() { //function to
run when exiting program
  led.writeSync(1);
  // 資源の解放
  led.unexport();
  buttonSw.unexport();
  console.log('program exit.');
}

// Ctrl-Cを入力して中断した時に呼び出される
// 関数の登録
process.on('SIGINT', unexportOnClose);
```

```
# LEDを点灯(終了するときは Ctrl-Cを押す)
$ node led_onoff3.js
```

イベント駆動

- 基本的にプログラムは上から書いた順に実行される
 - わかりやすいが複数の処理を効率的に行うには向きないことがある
 - 特に時間がかかる処理が含まれる場合, それ以降のプログラム全てに影響が出る
- 必要なタイミングにだけ必要なことを行うように, プログラムを記述することが重要
 - つまり, ある事柄(イベント)が発生した時に, プログラムを開始(駆動)される, ように作る
 - イベント: ネットワークに接続(or 切断)した, ファイルの読み込みが完了した, 時間が経過した, など

時間経過でLEDチカチカ

- タイマ処理を追加
 - 指定した時間が経過した時に関数を実行する
 - ここでは250ms毎にLEDの点灯消灯を繰り返し、5秒経過で終了する
- キーワード
 - setInterval, clearInterval
 - 周期処理の登録/解除
 - setTimeout
 - タイムアウト処理の登録

```
// タイマでLEDを点滅(led_onoff4.js)
var gpio = require('onoff').Gpio;
var led = new gpio(21, 'out');
var buttonSw = new gpio(13, 'in');

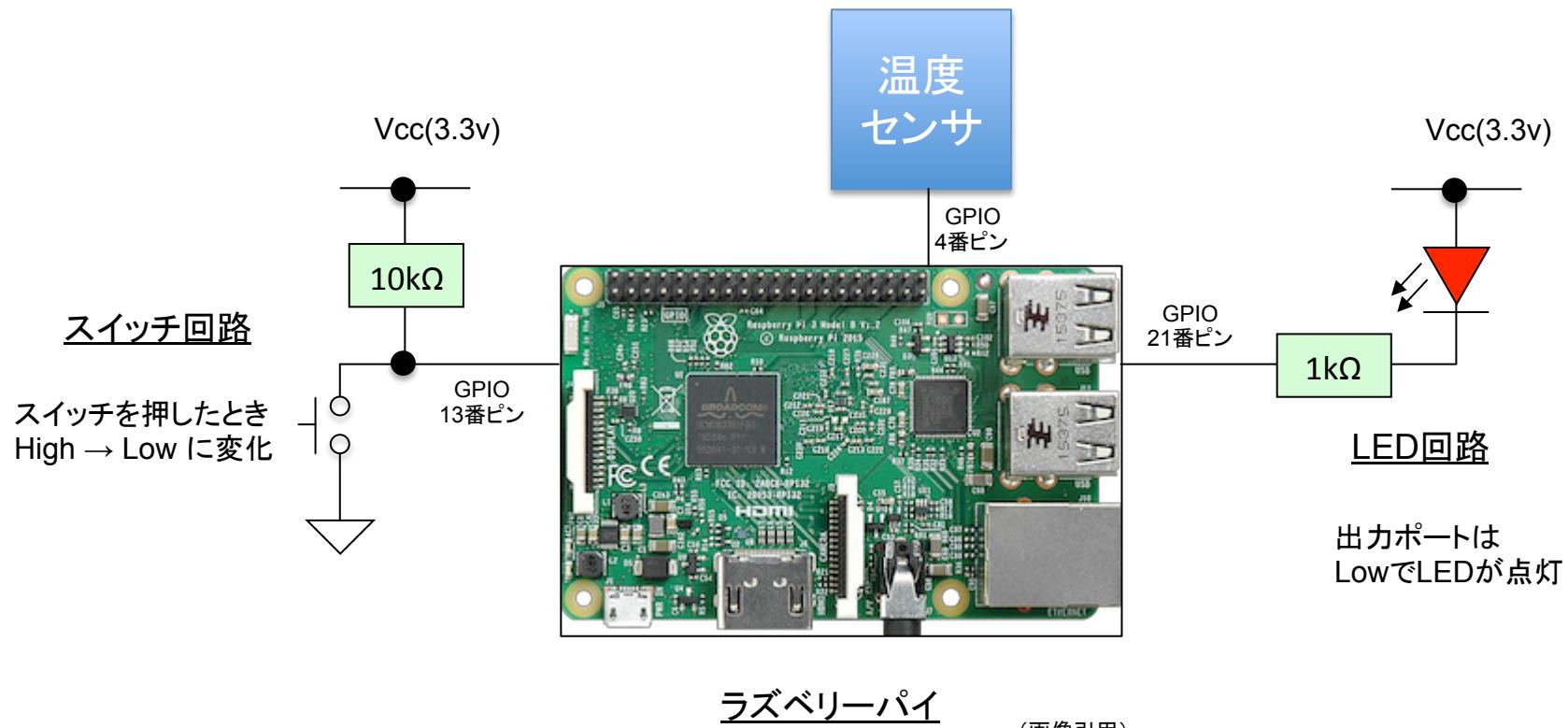
function blinkLED() {
  if (led.readSync() === 0) {
    led.writeSync(1);
  } else {
    led.writeSync(0);
  }
}
// 250ms経過毎に blinkLEDを呼び出す
var blinkInterval = setInterval(blinkLED, 250);

// 終了処理(点滅を止めて、LEDを消灯)
function endBlink() {
  clearInterval(blinkInterval);
  led.writeSync(1);
  led.unexport();
}

// 5秒後に endBlink を呼び出す
setTimeout(endBlink, 5000);
```

実習1-2: 温度測定

- 温度センサの値を取り込んでみる



(画像引用)
[https://en.m.wikipedia.org/wiki/
File:Raspberry_Pi_3_Model_B.png](https://en.m.wikipedia.org/wiki/File:Raspberry_Pi_3_Model_B.png)

node-dht-sensorモジュールを利用した 温度・湿度データの取り込み

- node-dht-sensor
 - 温度センサDHT{11, 22}からデータを取り出すモジュール
 - <https://www.npmjs.com/package/node-dht-sensor>
- キーワード
 - node-dht-sensor の read 関数
 - センサデータが読み出されたタイミングで、指定した関数が呼ばれる

```
// node-dht-sensorモジュールの読み込み
// (dht11_read.js)
var sensor = require('node-dht-sensor');

// センサーからの読み出し
sensor.read(11, 4, function(err, temperature, humidity) {
    if (!err) {
        console.log('temp: ' + temperature.toFixed(1) + '° C, ' +
                    'humidity: ' + humidity.toFixed(1) + '%'
        );
    }
});
```

課題

- 各自分で考えてみてください
 - スイッチを押したら、温度センサの値を測定してコンソールに表示する。
 - スイッチの状態変化に関するコールバック処理(割込み処理)に温度測定の処理を入れてみる
 - 一定間隔(例えば10秒)で温度、湿度を計測してコンソールに表示するプログラムを作る
 - setInterval 関数を使ってみる

実習1-3: カメラからの撮影

- カメラを用いて画像を撮影する
 - ここでは node-raspi-still モジュール(次項)を利用して取り込んでみる
- 課題
 - 温度センサの時と同様に、以下を考えてみてください
 - スイッチを押したら、カメラで画像をキャプチャし、ファイルに保存する。
 - 一定間隔(例えば1分)でカメラ画像をキャプチャし、ファイルに保存する

node-raspistillを利用した カメラ画像の取り込み

- node-raspistill
 - カメラからデータを取り出すためのモジュール
 - <https://www.npmjs.com/package/node-raspistill>
 - 内部で raspistill を利用しており、それをラップしたもの
- キーワード
 - takePhoto 関数
 - 画像を取り込む。既定では photo ディレクトリに画像が保存される

```
// カメラキャプチャサンプル(camera.js)
const Raspistill = require('node-raspistill').Raspistill;
const camera = new Raspistill();

camera.takePhoto()
  .then((photo) => {
    console.log('took photo', photo);
  })
  .catch((error) => {
    console.error('something bad happened', error);
  });
}
```

興味のある人は

- 他のセンサを試してみる
 - 音, 光, 温度, 傾き, 磁気, 水, などなど
 - どのセンサも最終的には電気信号に変換されるため,
GPIOに接続することでスイッチと同様に扱うことができます.

実習2-1:M2Xへの温度データ送受信

- やりたいこと
 - 温度センサの値をM2Xに送信し、データを収集する
 - M2Xに格納した温度センサの値をダウンロードする
- 方法:m2xモジュールを利用する
 - ドキュメント
 - <https://github.com/attm2x/m2x-nodejs/blob/master/README.md>
 - サンプル
 - <https://github.com/attm2x/m2x-nodejs/tree/master/examples>
 - <https://github.com/attm2x/m2x-nodejs#examples>



M2Xを利用する手順

1. M2Xのアカウントを作成し、ログインする
 - <https://m2x.att.com>
 - 説明は省略。アカウントの情報は当日説明
2. アカウントのマスターキーを確認する
3. マスターキーを元にNode.jsのコードを作成する
 - a. 新規デバイスとそれに対するストリームを作る
 - 最初に1度だけ
 - プログラムで作らずにM2Xのサイトで作成してもよい
 - b. 作成済みのデバイスおよびストリームを調べ、ストリームにデータを送信する
 - 毎回同じデバイスIDを調べるのは無駄なのであらかじめデバイスIDなどを調べてから送信しても良い

手順2: マスターキーの確認

The diagram illustrates the process of confirming a Master Key. It consists of two parts:

- Left Screenshot (Account Settings):** Shows the account navigation menu. The "Account Settings" option is highlighted with a red dashed box.
- Right Screenshot (Master Keys):** Shows the "Master Keys" table. A yellow-red redacted API key is shown in the "API KEY" column. The entire row is highlighted with a red dashed box.

A large blue curved arrow points from the left screenshot to the right screenshot, indicating the flow of the process.

手順3-a: デバイスとストリームの生成

```
// モジュールの取り込み
var M2X = require("m2x");

// マスターキーからM2Xオブジェクト取得
var m2xClient = new M2X("<API-KEY>");

// デバイス生成のためのパラメータ
device_params = {
    // デバイス名(必須)
    name: "RPi-Node",
    // visibility(必須): private or public
    visibility: "private",
};

// デバイス生成
m2xClient.devices.create(device_params,
function(response) {
    if (response.isSuccess()) { // 成功
        device_id = response.json.id;
        console.log("Device created.
Device id: ".concat(device_id));
    }
});

// ストリーム生成のための情報
stream_id = "temp";
stream_params = {
    // type: データ種別
    type: "numeric"
};
```

```
// ストリーム生成
m2xClient.devices.updateStream(
    device_id, stream_id, stream_params,
    function(response) {
        if (response.isSuccess()) { // 成功
            console.log("Stream created.
Stream id:".concat(stream_id));
        } else { // 失敗
            console.log(JSON.stringify(
                response.error()));
        }
    });
} else { // デバイス生成失敗
    console.log(JSON.stringify(
        response.error()));
}
});
```

```
# 実行結果
$ node m2x_create_device.js
Device created. Device id:
<生成したデバイスのID>
Stream created. Stream id: temp
```

手順3-b: デバイスの検索

```
// 既存のデバイスおよびストリームにデータを送信(m2x_search_device.js)

var M2X = require("m2x");
var m2xClient = new M2X("<API-KEY>");

// デバイスIDの検索(デバイス名は既知とする)
device_name = "RPi-Node";
device_id = "";

// 問い合わせる値(名前)
params = { name: device_name };

// デバイスの検索
m2xClient.devices.search(params, function(response) {
  if (response.isSuccess()) {
    devices = response.json.devices;
    for (var device_index in devices) {
      device = devices[device_index];
      if (device.name == device_name) {
        device_id = device.id;
        console.log("Found. Dev ID:".concat(device_id));
      }
    }
  } else { // search失敗
    console.log(JSON.stringify(response.error()));
  }
});
```

実行結果
\$ node m2x_search_device.js
Found. Dev ID:<生成したデバイスのID>

手順3-c: ストリームへのデータ送信

```
// 既存のデバイスおよびストリームにデータを送信(m2x_post_data.js)
var M2X = require("m2x");
var m2xClient = new M2X("<API-KEY>");

// デバイスID
device_id = "<デバイスID>";

// センサーからの読み出しと送信
var sensor = require('node-dht-sensor');
sensor.read(11, 4, function(err, temperature, humidity) {
  if (!err) {
    var temp = temperature.toFixed(1);

    console.log('temp: ' + temp + ', humidity: ' + humidity.toFixed(1));

    // 送信データの作成
    var params = { values: { 'temp': temp} };

    // データをポスト
    m2xClient.devices.postUpdate(device_id, params, function(response) {
      if (response.isSuccess()) { // 送信成功
        console.log(response.json);
      } else {
        console.log(JSON.stringify(response.error));
      }
    });
  }
});
```

```
# 実行結果
$ node m2x_post_data.js
temp: 18.0, humidity: 70.0
{ status: 'accepted' }
```

動作を確認

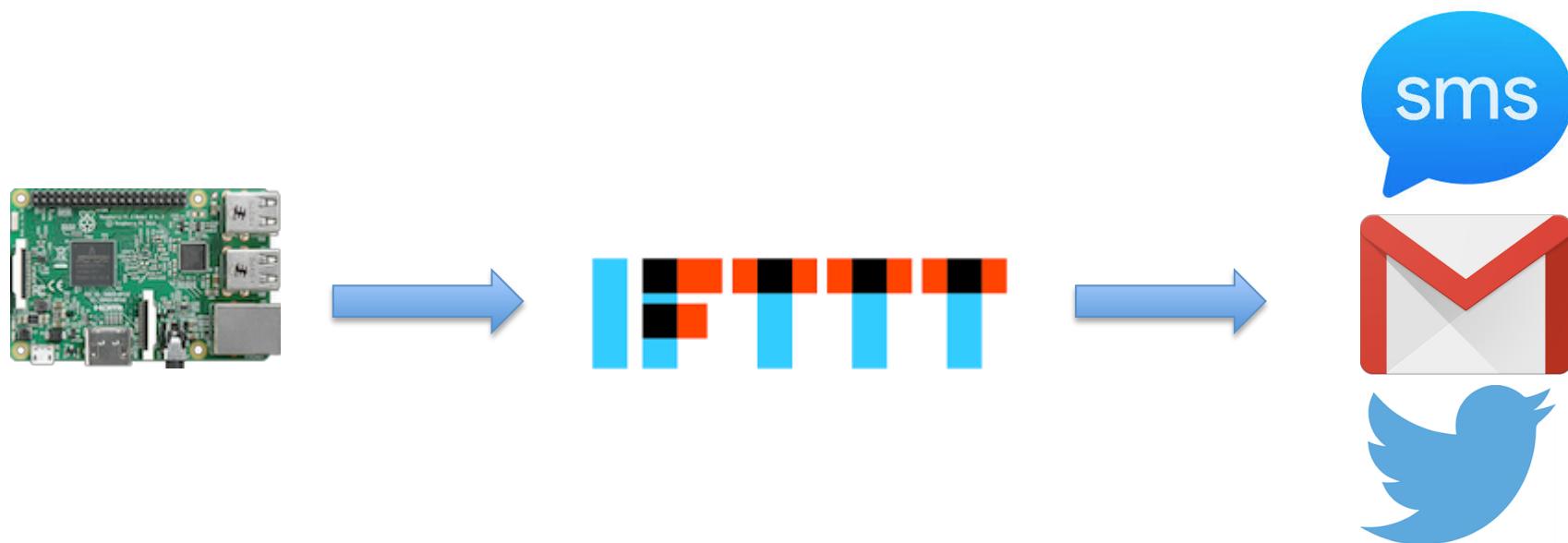
- データがM2Xに格納されているか
- グラフが表示されているか
- うまく動かない場合
 - アクセスキーは正しいか
 - デバイスID, ストリーム名は正しいか
 - その他, 何らかの通信エラーが発生していないか

興味のある人は

- データを一定の時間間隔で送信するようにしてみる
 - 例えば1分
- 温度に加えて湿度データも送ってみる
- M2Xからデータをエクスポートしてみる
 - 参考
 - https://github.com/attm2x/m2x-nodejs/blob/master/examples/export_values.js

実習2-2: IFTTTを利用した ネットワークサービスの連携

- イベント発生の通知やセンサの値などを、IFTTT(イフト)を経由して各種のネットワークサービスに送信する



IFTTTを利用する手順

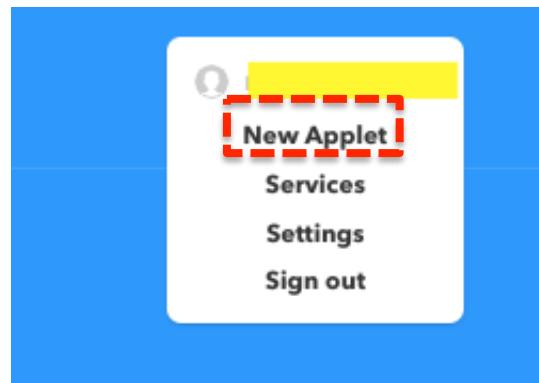
1. IFTTTのアカウントを作り、ログインする
 - <https://ifttt.com/>
 - アカウントの情報は当日説明
2. IFTTTでアプレットを作る
 - イベントID(Event ID)を覚えておく
3. プログラムを作る
 - イベントID(Event ID)を一致させる
4. 動かしてみる

IFTTTでアプレット作成

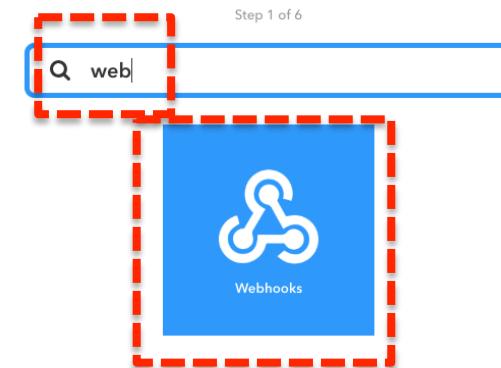
ログイン後、右上メニューから「New Applet」を選択

「This」を押す

Webhook選択

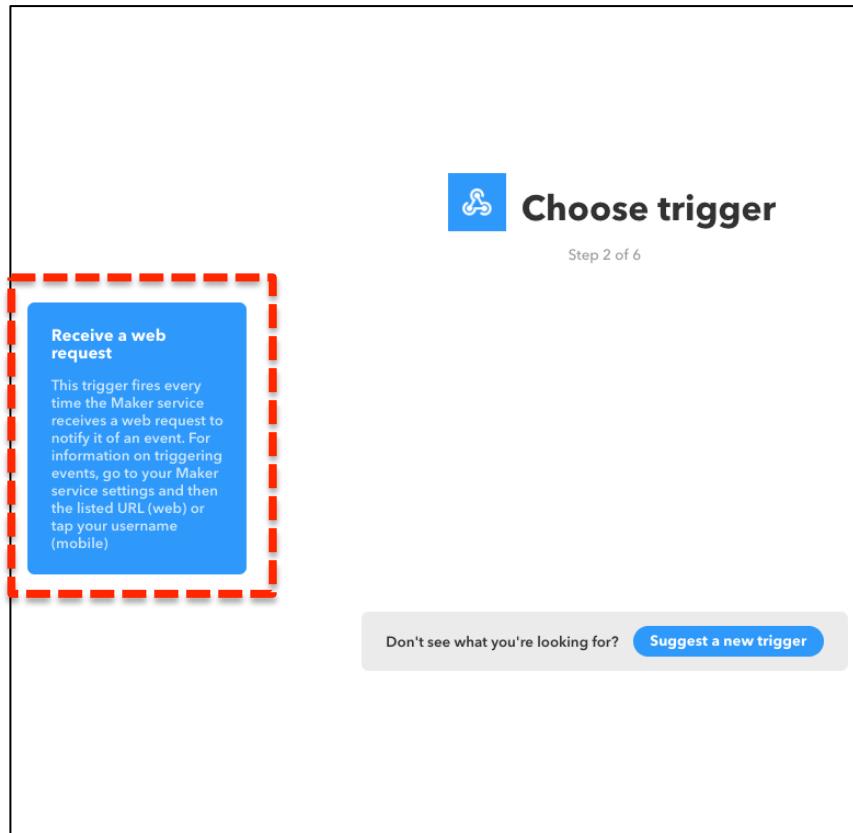


検索して候補を絞り込む **Choose a service**

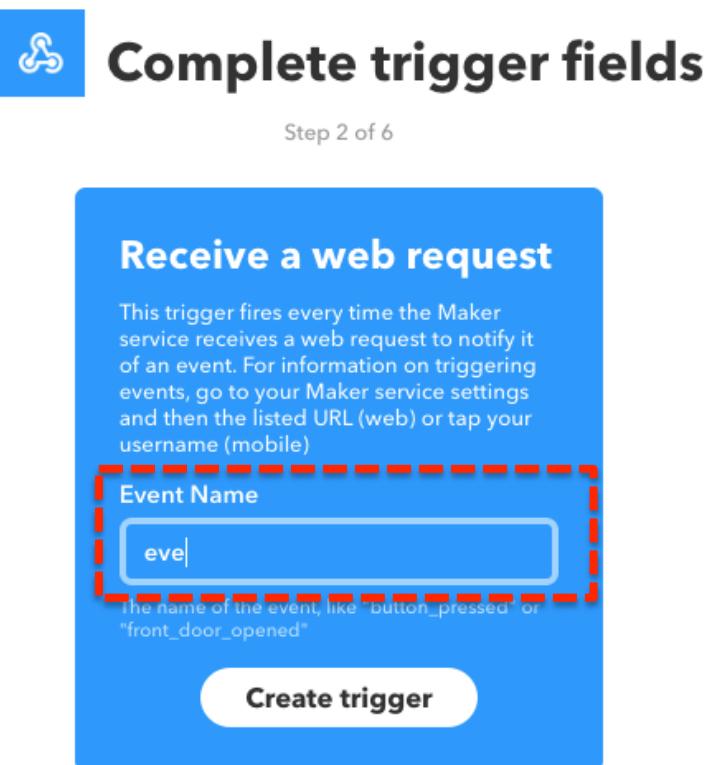


IFTTTでアプレット作成

トリガを選ぶ



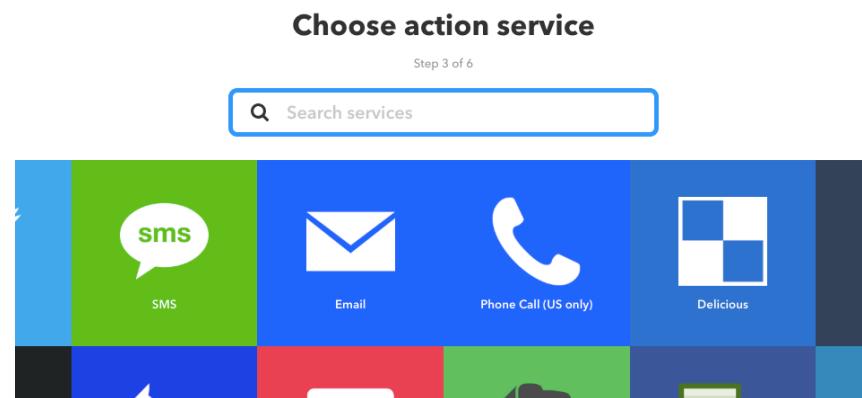
イベント名を決める



IFTTTでアプレット作成

「That」を押す

アクションを選ぶ



どのサービスを選ぶか

- 利用したいサービスに応じて選ぶ



たとえば、SMSと連携する場合

「Connect」を押す



Connect SMS

Step 3 of 6

Get important notifications on your phone via SMS. This service has a cap of 100 SMS messages per month for users in the US and Canada and 10 per month for those outside of North America. To avoid having Applets paused until the next month if you hit the limit, try the Notifications service. Some carriers outside of the US are not supported yet.

Connect

電話番号を入力し
PINコード認証



Connect SMS

Enter the phone number you would like to use for all your SMS Applets. For non-US numbers, include the leading 00 and country code. Not all international numbers may be supported.

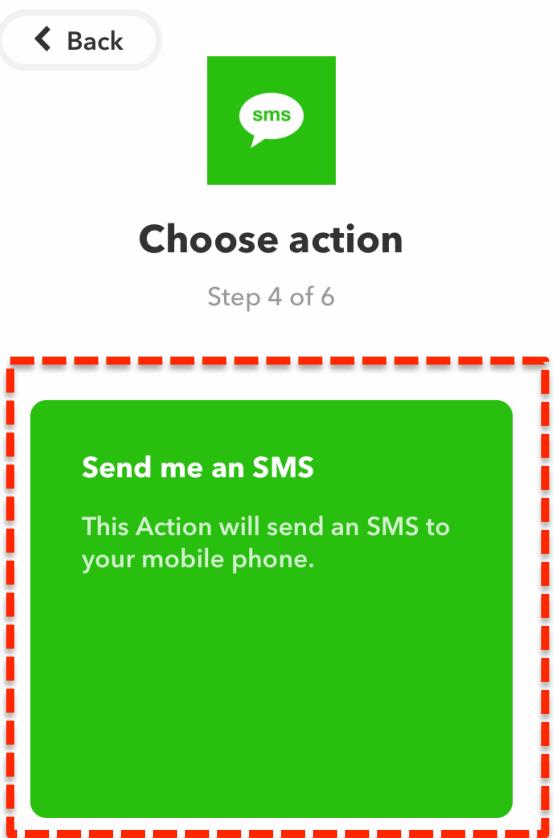
Your phone number

Send PIN

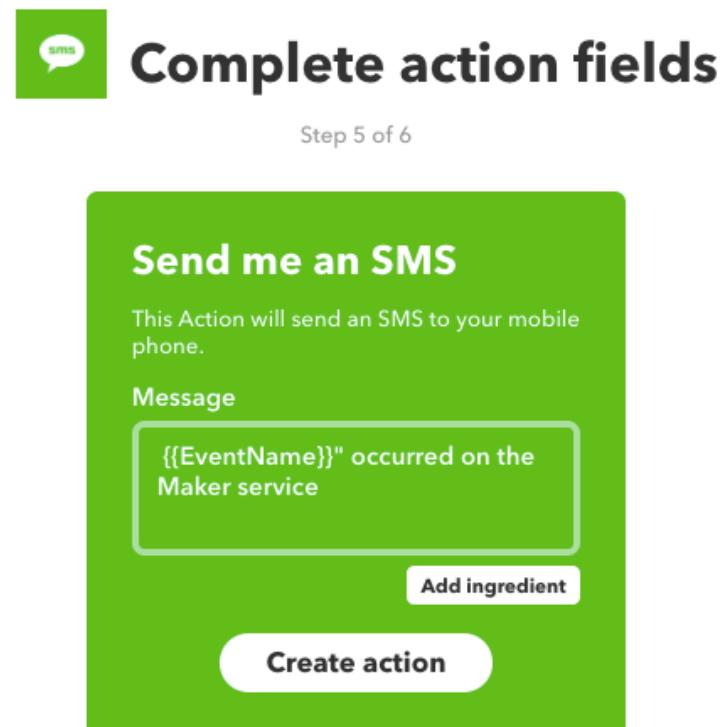
- 電話番号は「国番号(日本:0081)+電話番号」
- 例えば「090-1234-5678」の場合「008109012345678」となる

SMSと連携する場合

action を選ぶ



SMSで送られるメッセージを入力してcreate actionを押す



SMSと連携する場合

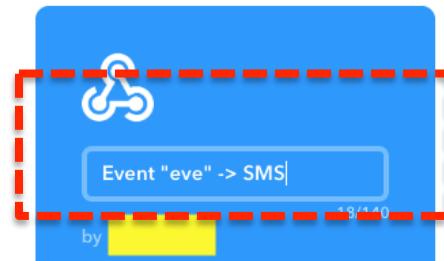
完了(メッセージを変更してお
くとあとでログを確認しやす
い)

完了後の様子

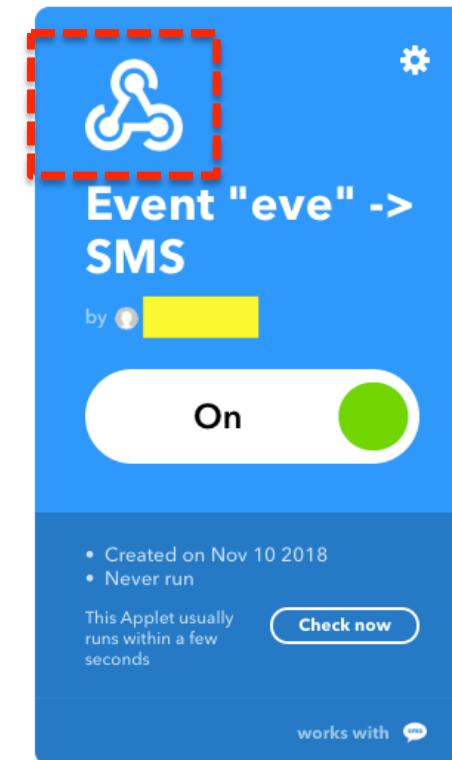
このあたりをクリックする

Review and finish

Step 6 of 6

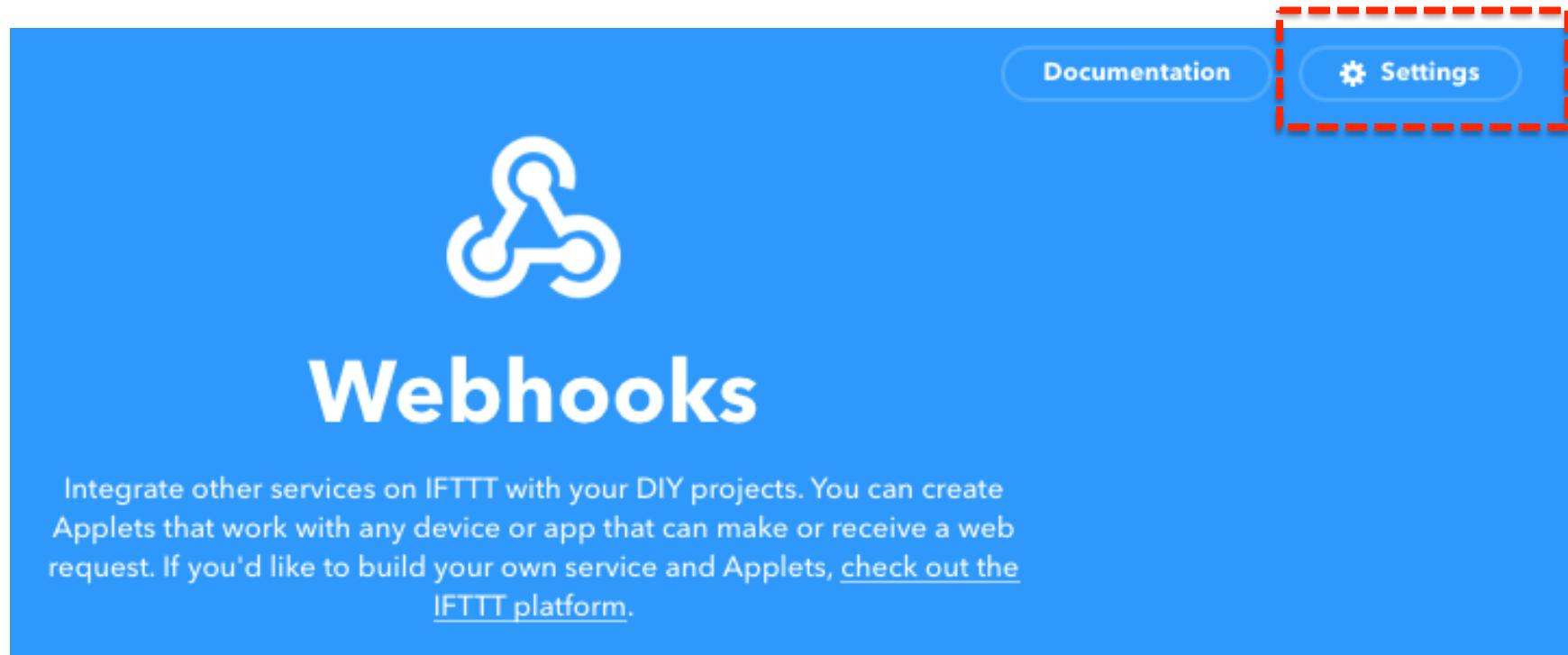


Finish



Webhookの設定確認

- プログラムでアクセスするURLを調べるため、
Webhookの設定画面へ移る
Settingsを押す



Webhookの設定確認

URL確認

The screenshot shows the IFTTT Webhooks settings page. At the top is a blue icon with three circles. Below it is the heading "Webhooks settings". A link "View activity log" is visible. Under "Account Info", it says "Connected as: [REDACTED]" and "Status: active". A URL "https://maker.ifttt.com/use/" is listed, with a red dashed box around it and the text "このURLにブラウザでアクセス" (Access this URL with a browser) written below it. A button "Edit connection" is also present. At the bottom, there's a "Disconnect Webhooks" link.

WebブラウザでURLにアクセス

The screenshot shows the IFTTT service page. At the top is a blue icon with three circles. Below it is the text "Your key is: [REDACTED]" with a "Back to service" link. In the center, there's a section titled "To trigger an Event" with the sub-instruction "Make a POST or GET web request to:". A URL "https://maker.ifttt.com/trigger/{event}/with/key/[REDACTED]" is shown in a text input field. A red dashed box surrounds this URL input field, with the text "このURLを使う" (Use this URL) written above it. Below the URL input is a JSON body example: "{ \"value1\" : \"[REDACTED]\", \"value2\" : \"[REDACTED]\", \"value3\" : \"[REDACTED]\" }". A note states: "The data is completely optional, and you can also pass value1, value2, and value3 as query parameters or form variables. This content will be passed on to the Action in your Recipe." Further down, it says "You can also try it with curl from a command line." followed by a "curl" command example. At the bottom right is a "Test It" button.

requestモジュールによる IFTTTへアクセスプログラムの作成

- requestモジュール
 - HTTPクライアントプログラムを作成するためのモジュール
<https://github.com/request/request>
<https://www.npmjs.com/package/request>

```
var request = require("request");

// IFTTT のイベントを駆動
request('https://maker.ifttt.com/trigger/<イベント名>/with/key/<Webhookのキー>',
  function (error, response, body) {
    console.log('error:', error);
    console.log('statusCode:', response && response.statusCode);
    console.log('body:', body);
});
```

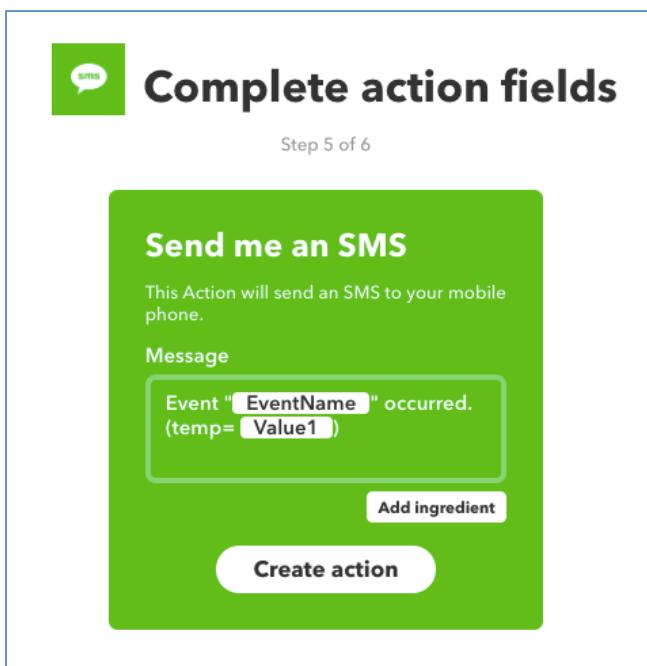
```
# 実行結果
$ node ifttt_1.js
error: null
statusCode: 200
body: Congratulations! You've fired the eve event
```

動作確認

- プログラムを実行したら、SMSを受信するかどうか
- うまく動かない場合
 - 電話番号が正しいか
 - イベントIDが正しいか
 - キーの値は正しいか

興味がある人は

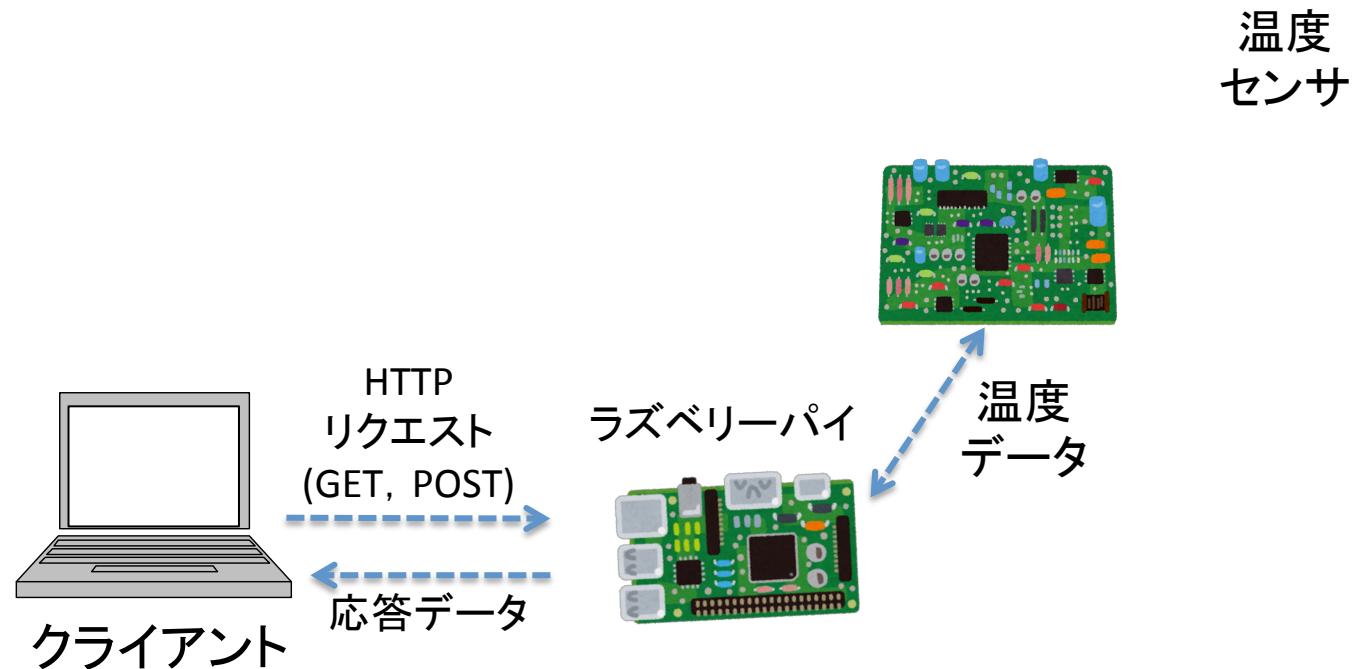
- 他のネットワークサービスも試してみる
- SMSメッセージにセンサのデータを入れてみる
 - POSTメソッドでアクセスする



ヒント: value1というデータに
温度データを入れてPOSTすると,
アプリett作成の際に「Add ingredient」で
「Value1」を選択して値を取り出すことができる

実習3: Webサーバの構築

- ルートパスへのアクセスでメッセージを表示する
- 温度センサの値を表示する



expressモジュールを使った HTTPサーバ版Hello World

- expressモジュール
 - <https://expressjs.com/ja/>
- Webブラウザで `http://<ラズベリーパイのIPアドレス>:3000/` にアクセスすると「Hello World」を表示
 - サーバ側は接続してきた相手のアドレスを表示

```
# 実行結果
$ node http_hello.js
server start
accessed to "/" from::ffff:192.168.0.10
```

```
// express を用いたHTTP版 Hello Worldサンプル
// (http_hello.js)
// Webブラウザで http://<ラズベリーパイのIPアドレス>:3000/ にアクセスすると Hello World を表示

// expressの初期化
var express = require('express');
var app = express();

// ルートパス (/)へのGETアクセスを受け取った時,
// メッセージを表示する
app.get('/', function (req, res) {
  console.log('accessed to "/" from' + req.ip);
  res.send('Hello World!');
});

// 3000番ポートで待ち受け
app.listen(3000, function () {
  console.log('server start');
});
```

温度センサの表示

- Webブラウザで `http://<ラズベリーパイのIPアドレス>:3000/temp` にアクセスしたとき、温度センサの値を表示するように追加する

```
// express を用いたHTTP版 Hello Worldサンプル(http_temp.js)
var express = require('express');
// expressの初期化
var app = express();

// ルートパス(/)にアクセスした場合の処理
app.get('/', function (req, res) {
  console.log('accessed to ' +
    req.originalUrl + ' from' + req.ip);
  res.send('Hello World!');
});
```

```
// /temp にアクセスしたら、温度センサーの値を送る
app.get('/temp', function (req, res) {
  console.log('accessed to ' +
    req.originalUrl + ' from' + req.ip);
  // 温度センサーからの読み出し
  var sensor = require('node-dht-sensor');
  sensor.read(11, 4, function(err, temperature,
    humidity) {
    if (!err) {
      res.send(temperature.toFixed(1));
    }
  });
  // 3000番ポートで待ち受け
  app.listen(3000, function () {
    console.log('server start');
  });
});
```

動作確認

- アクセスした時にちゃんとデータが表示されるか確認する



興味のある人は

- 他のデータ、例えば湿度データやカメラ画像なども同様に表示できるよう変更してみる
 - 画像の場合は、一旦ファイルに保存しておいてからそれを表示するHTMLデータを生成してもよい

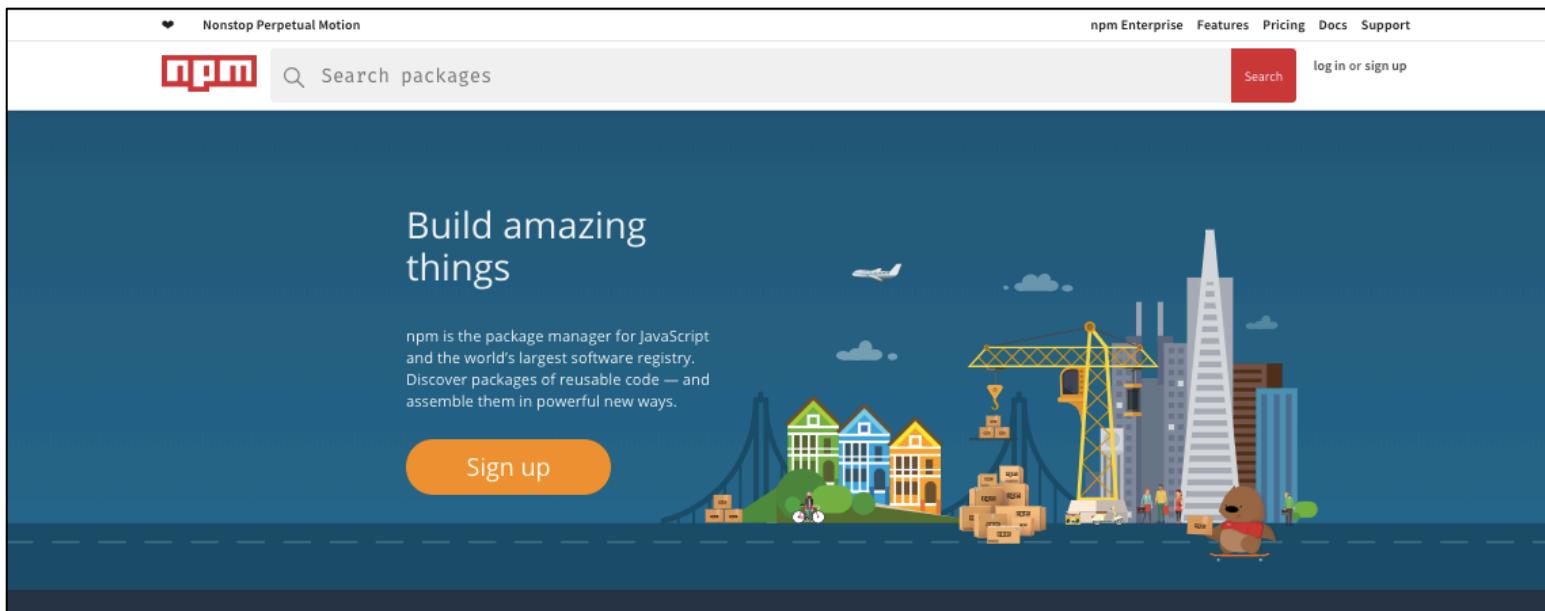
その他の話題

デーモン化

- サービスとしてバックグラウンドで動かし続ける場合
- forever
 - <https://github.com/foreverjs/forever>
- pm2
 - <https://github.com/Unitech/pm2>

パッケージ管理

- 必要に応じパッケージを検索し、必要なパッケージをインストールする必要がでてくる
- npm (node package manager)
 - ライブラリパッケージのインストールおよび削除
 - 公式サイトでパッケージの検索が可能
 - <https://www.npmjs.com/>



ネイティブ拡張

- C/C++のライブラリをNode.jsから利用する
 - Node.jsでサポートされていない機能を利用する場合など
- 参考
 - C++ Addons (Node.js v11.3.0 Documentation)
 - <https://nodejs.org/api/addons.html>
 - Native Abstractions for Node.js
 - <https://github.com/nodejs/nan>

まとめ

- ラズベリーパイでNode.jsを利用した簡単なプログラムを作成した
 - デバイスの制御
 - スイッチ, LED, 温度センサ, カメラ
 - クラウドへの通知やデータ送信
 - M2X, IFTTT
 - Webサーバの構築によるデータの表示
- この他にもNode.jsは様々な利用方法がありますので、興味がありましたらいろいろ試してみてください
 - 質問なども受け付けています。