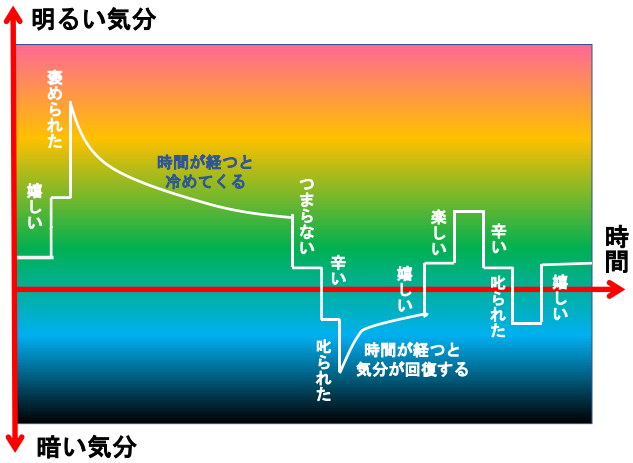
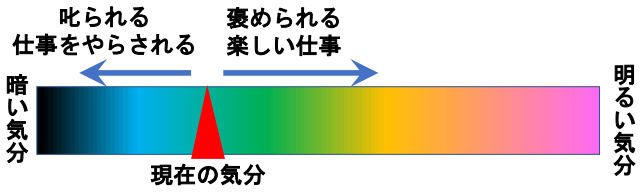
Raspberry Pi 中の上級電子工作

対話型秘書ロボット

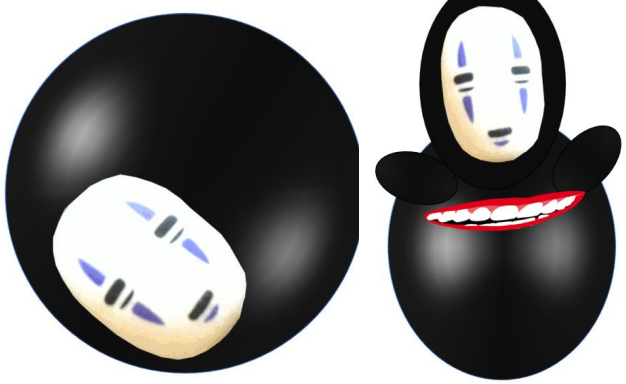
マルチプロセス・マルチプロセッサ化へのアプローチ





著者　秋山忠次

（非売品）



目次

はじめに.....1
1 Raspberry Pi プロジェクト（第 2 号）秘書ロボット.....2
1.1 電子工作「入門」を卒業したら.....2
1.2 開発のアプローチ.....2
1.3 ロボットの仕様構想を描く.....3
1.4 業務に分類する.....3
1.5 ロボットの外形を構想する.....4
1.6 ソフトウエア開発にむけて.....4
1.7 プロジェクトに必要な機材.....5
2 開発計画と仕様設計.....6
2.1 作業分析.....6
2.2 Value Engineering.....7
2.2.1 機能ごとのコスト分析と代替案検討.8
2.2.2 代替案の検討.....8
2.2.3 Web サーバー.....8
2.3 段階的開発アプローチ.....9
2.5 開発仕様.....11
2.6 要素技術.....11
2.7 サブシステム構成.....12
3 ハードウエアの準備.....13
3.1 必要なハードウエア.....13
3.2 Raspberry Pi ZERO WH の準備.....15
3.2.1 Linux のインストール.....15
3.2.2 Raspberry Pi の基本的な設定.....16
3.2.3 WiFi の設定.....16
3.2.4 ディレクトリ構造.....17
3.2.5 ソフトウエアパッケージのインストール.....17
3.3 オーディオユニット.....20
3.3.1 音声再生.....21
3.3.2 音声録音.....21
3.3.3 音量の調整.....21
3.4 カメラユニット.....22
3.4.1 静止画撮影.....22
3.4.2 動画撮影.....22
3.5 LED ユニット.....23
3.6 電源ユニット.....25
3.7 ロボット本体の組み立て.....25
3.8 全回路図.....26
4 既存ソフトウエアの組み込み.....27
4.1 音声合成.....27
4.1.1 Open JTalk.....27
4.1.2 クラウドサービス.....29
4.2 音声認識.....29
4.2.1 Julius.....29
4.2.2 クラウドサービス.....30
5 ソフトウエア開発環境.....31
5.1 ソフトウエアの部品化.....31
5.2 Linux ツールの利用.....31
5.2.1 cpp.....31
5.2.2 リダイレクトとパイプ.....33
5.2.3 クリーンアップ.....34
5.2.4 シェルスクリプト.....34
5.3 モジュール化.....35
5.4 プログラムの検証.....35
5.4.1 シミュレーション用コード.....35
5.4.2 デバグ用コード.....35
5.4.3 検証用代替プログラム（スタブ）.35
6 システム構成の検討.....36
6.1 マルチプロセスシステム.....36
6.2 プロセス間通信.....36
6.2.1 TCP/IP 通信.....36
6.2.2 FIFO 通信.....37
6.2.3 シグナル.....37
6.3 定型音声ファイル.....37
6.4 音声コマンド辞書.....38
6.5 人工知能の実装方法.....40
7 プロトタイプの設計とサブシステムの検証 41
7.1 サブシステムとプロセスの設計.....41
7.1.1 聴覚サブシステム.....41
7.1.2 発話サブシステム.....42
7.1.3 タイマーサブシステム.....43
7.1.4 Web サーバー.....43
7.1.5 AI エンジン.....44
7.2 共通ヘッダーファイル.....44
7.3 聴覚サブシステム.....47
7.3.1 Julius の常駐化.....47
7.3.2 音声入力サーバーの設計.....47
7.3.3 モジュールモード Julius 単体検証.49
7.3.4 音声入力サーバー単体検証.....49
7.3.5 聴覚サブシステムの検証.....50
7.4 発話サブシステム.....53
7.4.1 音声再生クライアント.....53
7.4.2 Open JTalk の常駐化.....56
7.4.3 サブシステム検証.....59
7.5 タイマーサブシステム.....60
7.6 Web サーバー.....61
7.6.1 共通インクルードファイル.....61
7.6.2 Web サーバー.....62
7.6.3 favicon.ico.....63
7.6.4 HTML 文書.....63
7.6.5 検証.....64
7.7 AI エンジン.....64
7.7.1 感情表現モジュール emotion.py.....66
7.7.2 時間取得モジュール get_time.py.....67
7.7.3 音声処理モジュール mouth.py.....68
7.7.4 時間管理 モジュール time_keeper.py.....71
7.7.5 挨拶モジュール greeting.py.....74

7.7.6 命令解釈モジュール interpreter.py ..75
7.7.7 AI エンジン冒頭部　kaonashi.py 77
8　プロトタイプの評価と実証モデルに向けての検討 79
8.1　総合検証 79
8.1.1 サブシステムの立ち上げ 79
8.1.2 ブラウザからの命令 79
8.1.3 音声命令 79
8.2　パフォーマンス評価 79
8.3　実証モデルにむけた改善項目 80
8.3.1 音声認識の処理時間短縮 80
8.3.2 感情表現の改善 81
8.3.3 カオナシ Web 画面の会話履歴 82
8.4　実証モデルで追加する機能の検討 83
8.4.1 挨拶の充実 84
8.4.2 画像取得 84
8.4.3 画像表示 84
8.4.4 人物認証、名前管理、人物情報 85
8.4.5 調査・検索、天気照会 85
8.4.6 音楽などの mp3 ファイルの再生 86
8.4.7 時刻管理 86
8.4.8 検索クラウドサービス 86
9　実証モデルの開発 87
9.1　システム構成 87
9.1.1 マルチプロセッサシステム設計上の注意点 87
9.1.2 プロセス ID を得るために 87
9.1.3 子プロセスへの指令 88
9.1.4 個人情報の一括管理 88
9.2　聴覚サブシステム 88
9.2.1 インクルードファイル 88
9.2.2 音声入力サーバー 88
9.3　発話サブシステム 90
9.3.1 音声処理モジュール mouth.py 90
9.3.2 音声出力サーバー 90
9.4　視覚サブシステム 91
9.4.1 インクルードファイル 91
9.4.2 画像取得モジュール 92
9.4.3 画像入力プロセス 92
9.5　メール送信サブシステム 93
9.5.1 インクルードファイル 93
9.5.2 メール送信モジュール 93
9.6　タイマーサブシステム 94
9.6.1 時間管理モジュール 94
9.6.2 時間取得モジュール 94
9.7　Web サーバー 94
9.8　感情サブシステム 94
9.8.1 インクルードファイル 94
9.8.2 感情処理モジュール 95
9.8.3 感情表現プロセス 95
9.9　検索サブシステム 95
9.9.1 ライブラリのインストール 95
9.9.2 JSON 95
9.9.3 Google Assistant API 96
9.9.4 Google への登録 97
9.9.5 インクルードファイル 97
9.9.6 検索実行モジュール 98
9.9.7 検証 99
9.9.8 検索応答モジュール 99
9.10　AI エンジン 100
9.10.1 挨拶モジュール 100
9.10.2 命令解釈モジュール 100
9.10.3 AI エンジン冒頭部 101
9.10.4 個別検証 101
9.10.5 自動起動 101
10　実証モデルの評価とフルモデル開発に向けて　102
10.1　実証モデルの評価 102
10.1.1 応答パフォーマンス 102
10.1.2 機能の充足度 102
10.1.3 既存品との比較 102
10.2　フルモデルの開発にむけて 104
10.2.1 高性能サーバーへの移植 104
10.2.2 インクリメンタル拡張 105
10.2.3 ルールエンジンの実現 105
10.3　フルモデルで拡張したい機能 105
10.3.1 会話履歴で変化する応答 105
10.3.2 応答の確認 105
10.3.3 雑談 106
10.3.4 隠し芸の充実 106
10.3.5 気まぐれ行動 106
10.3.6 検索機能の拡張 107
10.3.7 感情表現 107
10.4　フルモデルで新たに実装する機能　107
10.4.1 予定管理 107
10.4.2 メールの送受信 107
10.4.3 音声・画像通信 107
10.4.4 話相手の認識 108
10.4.5 感情の読み取り 108
10.4.6 機器操作 108
10.4.7 やっぱり C-3PO 109
10.4.8 可動型ロボット 109
11　プロジェクトを振り返って 110
11.1　実証モデルまでの成果 110
11.2　常識外れ（？）のアプローチ 110
付録 111
Linux と Windows のファイル 111
プロジェクトファイル 112

コラム一覧

人造人間こと始め……5
最初のロボット……9
ロボット工学の三原則……10
異星植民地の運営……11
その後の三原則……26
LISP と Prolog……39
ELIZA と話す……40
日本発の大規模プロジェクト……46
日本のロボット史（その 1）……52
日本のロボット史（その 2）……78
球形の牛の先は？……101
暗いロボット……103
プロジェクト・ゼロ……104
C-3PO のお祖母さん……107
ロボットのいない未来……112

はじめに

2014 年の秋、書店で出版されたばかりの一冊の本[1]を見つけました。読んですぐに、「これはいい！」と思いました。Raspberry Pi は、小型 CPU カードとして非常に使いやすいと感じたからです。その理由は、

1. すぐに OS が立ち上げられるよう、Raspbian が提供されている
2. emacs、cpp など、Linux のソフトウエア開発環境が活用できる
3. 自作ソフトウエアの大部分が Linux を搭載した PC 上でも走らせることができる
4. 周辺ハードウエアを駆動するライブラリが提供されているので、ハードウエアの詳細を知らなくても使える

などです。さっそく Raspberry Pi を手に入れ、最初のプロジェクトとして、調理用の温度コントローラを開発しました。仕事の合間に進めたので、実用になるまで 1 年以上かかりました。ずっと使ってきて、改造は今でも続いています。

2020 年になって、次のプロジェクトとして、秘書ロボットの開発に取り掛かることにしました。今度は、私自身の勉強も兼ねて、かなりソフトウエア中心のプロジェクトです。かなりの時間がかかると覚悟していたのですが、その頃「新型コロナウイルス」の感染が広がりました。外出自粛を余儀なくされ、思っていた以上に進んでしまったのです。

プロジェクトの内容を公開しようと、資料をまとめているうちに、思いがけないことに気が付きました。今回のアプローチは、だいぶ前に読んだ「Fear of Physics（物理学は怖いもの？）」という本[2]の考え方を踏襲していたということです。

あろうことか、早川書房から出ていた邦訳の書名は『物理学者はマルがお好き』というものです。冒頭で語られるのは、酪農場の業務改善について助言を求められた、工学者、心理学者、物理学者の話です。工学者と心理学者が、それぞれの視点から提案した後、最後に黒板の前に立った物理学者は、おもむろに、こう始めます「牛を球だとしてみましょう」。

私がこの本に惹かれたのは、多くの示唆があるからでした。それは、

A. 牛をまず球とみなせ（複雑なものは、枝葉末節を切り捨て、簡略化して考えよう）
B. 落とし物は街灯の下から探せ（できる可能性のあるところから手をつけよう）
C. うまく行ったら、また同じようにやれ（検証されたものは、何度でも使いまわそう）

などです。いずれも「社会常識」に反するようにも聞こえますが、実はとても有用なことでした。今回のプロジェクトについて読み替えてみれば、

A. いきなり複雑なシステムにしない（開発フェーズを設定し、発展させていく）
B. 実現に時間がかかりそうなものは後回し（簡単なお試しだけにしたり、延期したりする）
C. 実績のある手法は繰り返し使う（温度コントローラで役立った手法を再利用する）

となります。このおかげで、片意地を張ったりせず、少しずつ実現していくことを楽しむことができました。趣味としては、この方が本来の姿かもしれません。温度コントローラのときは、きっちりした仕様書を用意し、そのとおりに実現していきましたが、今回はかなり柔軟に取り組みました。この過程を追体験することで、なにか得るものがあればいいと思って、公開します。

なお、開発プロジェクトとは直接関係のない、趣味的な記事を、コラムとして掲載しています。楽しんでいただければ幸いです。

2020 年 8 月　著者

[1] 金丸隆志「Raspberry Pi で学ぶ電子工作 — 超小型コンピュータで電子回路を制御する」講談社ブルーバックス；　2016 年に「最新 Raspberry Pi で学ぶ電子工作 — 作って動かしてしくみがわかる」に改訂された。

[2] Lawrence M. Krauss (1993)著。邦訳初版は「物理の超発想（講談社, 1996）」。2004 年に早川書房から＜数理を楽しむ＞シリーズとして、改題再版。

1　Raspberry Pi プロジェクト（第 2 号）秘書ロボット

この本では Raspberry Pi を利用して、一種のコミュニケーションロボット（人と交信するロボット）を作る過程を、一つのプロジェクトとして紹介します。

前のプロジェクトは「中級」でした。今回のプロジェクトを「上級」と呼ぶのは、少しおこがましいと思い、控えめな表現である「中の上」級としました。前よりは難しかったので。

1.1　電子工作「入門」を卒業したら

Raspberry Pi を使った「電子工作入門」という書籍が多く出版されています。ひとむかし前の小型 CPU カードと比べると、Raspberry Pi は格段に扱いやすい素材です。とくに Linux という、ソフトウエア開発に適した環境が後押しをしてくれます。気楽に電子工作を始めるきっかけになるので、入門書を活用して大いに経験を積みましょう。

でも、「LED がチカチカした」とか「温度センサの表示が読めた」という『体験』まで行きついた先はどうでしょう？　もう少しまとまったものを作って、その手法と経験を学ぶことが、何かの役に立つかもしれません。プロジェクトの達成感と反省は、単に役に立つことより、もっと有意義なものになると思います。

私の最初のプロジェクトは、燻製などの調理に使える「温度コントローラ」でした。その経過と経験は『Raspberry Pi 中級電子工作：温度コントローラ～設計から製作・検証・応用・保守まで』にまとめてあります。そこでは、設計手法から検証方法、さらに機能拡張の方法を説明しています。どちらかと言えば、ハードウエアの製作と駆動が重要なテーマでした。その拡張はまだ続いていますが、もう少し上級に進むため、次のプロジェクトを構想しました。

最近は Raspberry Pi を使って、人工知能（AI）や機械学習を学ぶ人が増えています。Linux とさまざまなソフトウエアを利用すれば、面白いことができそうです。ただ、それなりの CPU パワーを必要とするソフトウエアが多いことも事実です。もう一つの潮流はロボット（動くロボット）でしょう。Raspberry Pi の GPIO、特にパルス幅変調（PWM）を使って、モーターを駆動すれば、動くロボットが実現できます。

しかし私の興味と志向は、省エネ小型サーバーにあります。とくに Raspberry Pi ZERO は消費電力も小さく、気軽に使えるプラットフォームです。ロボットの身体に埋め込むためには、小型で発熱の少ないことが必須条件だったのです。貧弱な CPU パワーで、どんなことまで実現できるのか試すのと、最近の技術動向を学ぶことを目的にしました。

目標設定にあたり、最初に参考にしたのは、「**スマートスピーカー**」です。音声で要求すると、何らかの返事や操作をしてくれるスピーカーで、Raspberry Pi 用のキットも販売されています。ただ、実際の AI 機能は、ほとんどクラウドサービスで実現しているので、勉強するところが少ないのが残念です。

次の参考は、いわゆる「**コミュニケーションロボット**」です。人と会話したり、言葉に反応したりするロボットで、「カワイイ」癒し系のロボットと、動き回るロボットがあります。一時期話題になった鉄腕アトムの組み立てキット（裏表紙）でも、Raspberry Pi を採用していました。

最後のヒントは、「**分身ロボット**」OriHime（https://orihime.orylab.com/）でした。一種の「**テレプレゼンス**」ロボットで、離れた場所にいるロボットを使って、対話や操作を行えます。ビデオ会議とロボットを組み合わせた機能と言えばいいでしょうか。この三つのアイディアをもとに、対話ができる「**秘書ロボット**」を、目標として選択しました。移動したり、腕を動かしたりする機能は、今回は封印し、その次のプロジェクトに取っておきます。

ロボットはある程度の期間、運用し続けることになります。その間に問題が起こったり、もっと良くしたいという欲が出てきたりします。そのためには、修理や改造といった保守（メンテナンス）を考慮する必要があります。保守を視野に入れた手法を採用したで、皆さんの参考になると良いのですが。

1.2　開発のアプローチ

秘書ロボットを作ることを目標に、開発に着手しました。このプロジェクトの特徴的なアプローチを説明します。

最初のプロジェクトである温度コントローラの場合は、開発する機能が明確です。まずは仕様書にまとめました。開発目標がはっきりすれば、コントロー

ラの構造やソフトウエアの設計を、ストレートな手順に従って進められます。上位設計から、下位の設計に順次ブレークダウンしていけました。しかし今回のプロジェクトでは、それとは違ったアプローチが必要です。

実現するための手段がいろいろ考えられ、調べたり試したりしないと、どれが最適なのか分かりません。試した結果、Raspberry Pi ZERO では CPU パワーが不足して、使い物にならないことも十分あり得ます。それで、前に採用したトップダウンの手順重視型アプローチではなく、研究型アプローチを取ることにしました。

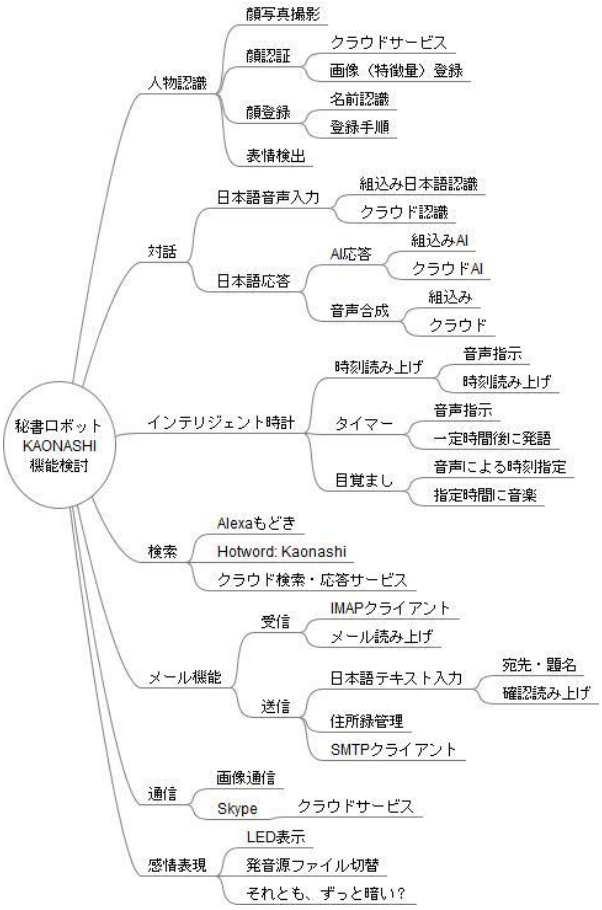
まず、開発したいもの（目標仕様）を厳密に決めたりせず、「これが実現できればいいな～っ」という程度の、ざっくりとした仕様（というか、構想あるいは夢想）にします。この段階では実現の手段や困難さを無視して、高めの目標を掲げます。次に、それを複数の機能に分解し、各々の機能を実現する方法を研究します。具体的には、(a) 自作する、(b) 既存の技術を組み込む、(c) クラウドサービスを利用するといったさまざまな手段を比較検討し、どの技術やサービスがどう使えるのかを考えます。実現のめどが立ったところで、もう一度開発仕様に戻って詳細なものにします。この段階で、目標の一部を切り捨てたり、後回し（拡張機能）にしたりして、実現可能なシステムを目指します。

開発目標を一度に実現しようとせず、一部の簡単な機能だけを実現したモデル（プロトタイプ）を、まず作り上げます。ただし、最初に全体構造（システムアーキテクチャ）を設計し、この段階からそれに従います。そうすることで、アーキテクチャの有効性を確認していきます。それから機能を拡張して、最終目標に近づけていきます。

プロトタイプでは、部品となる技術（要素技術）をすべて Raspberry Pi ZERO 上で実現しようと思います。要素技術に関する勉強をすることも、プロトタイプ製作の目的だからです。出来上がりを評価しながら、次のステップでは、一部の機能を他の CPU やサーバーに移すことを検討します。

1.3　ロボットの仕様構想を描く

秘書ロボットに搭載したい機能を、思いつくままに書き出し、マインドマップで体系化してみました。マップには実現手段を前提にした考察も含まれていますが、とにかくリストアップすることが先決です。これをもとに体系化と詳細化を行っていきます。



秘書ロボットにやらせたい作業（第一版）

1.4　業務に分類する

前節の構想をもとに、「どうやって作るか」は、いったん忘れ、「何を実現するか」という視点で、開発目標を整理します。

スマートスピーカー、コミュニケーションロボット、分身ロボットの機能を参考に、秘書ロボットにやらせたい「業務」（まだ妄想を含む）を、以下のように書き出しました。

業務分類	業務例
挨拶	業務開始、あいさつ、人物認証、業務終了（片付）
問い合わせ	時間やスケジュールの問い合わせ、調査・検索
事務作業	時間管理、機器操作、口述筆記
会議・通信	遠隔地の相手との会談、電話対応、メール処理
資料管理	人物に関する情報の管理、スケジュールの管理

自分の仕事を助けてくれる、人間の秘書の仕事の大半をカバーしていますが、お茶くみは次の機会に譲ります。業務分類ごとに、具体的な「作業」として記述していきます。

挨拶業務

作業	具体的な作業内容
業務開始	業務開始の挨拶をする
あいさつ	挨拶されたら、秘書からも挨拶する
人物認証	挨拶してきた人物を認識し、名前を呼ぶ
業務終了	業務終了の挨拶をしてシステムを終了する

問い合わせ業務

作業	具体的な作業内容
日付照会	問われたら今日の日付を答える
時刻照会	問われたら現在時刻を答える
天気照会	これからの天気予報を答える
ニュース照会	最新のニュースを伝える
予定照会	問われたら今日の予定を答える
調査検索	依頼された項目を調査（検索）して答える

事務作業業務

作業	具体的な作業内容
時間管理	指定した時間がたった教える
時刻管理	指定した時刻になったら教える
予定通知	予定の 5 分前に教える
機器操作	ステレオ、テレビ、ビデオ、空調、照明などを操作する
口述筆記	口述した文章をテキストファイルにする

会議･通信業務

作業	具体的な作業内容
電話開始	相手に音声通話を要求する
電話対応	音声通話要求を受け、指示を受けて接続する
会議開始	相手にビデオ通話を要求する
会議対応	ビデオ通話要求を受け、指示を受けて接続する
メール送信	口述筆記した文章をメールとして送信する
メール受信	受信したメールを読み上げる

資料管理業務

作業	具体的な作業内容
名前管理	教えられた名前を記録する
写真管理	写真を撮影し、指定した人物と対応付ける
人物情報	名前、写真、メールアドレス等の保存と取り出し
予定管理	予定の追加と削除を行い、一覧表を作成する

1.5　ロボットの外形を構想する

秘書ロボットと言っても、目に見えない「仮想ロボット」では面白くありません。やはり目に見える姿が大事です。

「秘書」というと、すっきりとしたスーツにハイヒールの美人秘書や、議員秘書のような背広姿を想像しますが、その印象を引きずりたくありません。

移動したり腕を動かしたりしないので、あまり外形に制約はありません。話しかけるのには顔があった方がいいので、人形かぬいぐるみが候補になります。どんな人形でも良いのですが、個人的な好みを優先し、アニメ「千と千尋の神隠し」に出てきた「カオナシ」を使うことにしました。身体が黒くて形態も自由なので、電子回路を内蔵させるのにも適しています。名前もカオナシにしました。

1.6　ソフトウエア開発にむけて

秘書ロボットの機能を作りこむ作業の大部分は、ソフトウエアの開発です。実用的なソフトウエアの開発を手掛けたことのある人なら、モジュールの仕様設計と実装設計、プログラムの作成、出来上がったモジュールの検証という三項目にかかる時間は、ほぼ同程度だという実感があると思います。場合によっては、プログラムの作成にかける時間がいちばん短いこともあります。この本に許される分量のせいで、検証の説明を貧弱なものにせざるを得ませんでしたが、別途ダウンロードできるプロジェクトファイルには、検証用のプログラムが多く含まれています。

Raspberry Pi で GPIO の操作をプログラムするのに便利な言語は Python です。この本でも、できる限り Python で記述するようにしています。サポート終了が近づいている Python2.7 ではなく、Python3 を使用しました。ただし Web サーバーの設計には JavaScript と HTML を、既存ソフトウエアの改造には C 言語を使う必要がありました。古いプログラミング教育のせいか、私の書くものは逐次手順になりがちで、必ずしも Python の機能をうまく引き出していないときがあります。ご指摘いただければ幸いです。

Python でプログラムを書いていると、インタープリターの便利さから、使い捨て同然の扱いをすることを、よく見かけます。しかし、ここではそれを避け、Linux に備わったツールを使ってソフトウエアの保守性を良くしています。あまり一般的ではない

かもしれませんが、とても役に立つ手法なので、あえて採用しました。

インターネット上では、Raspberry Pi に限らず、いろいろな製作（試作？）記事が見つかり、参考になるものが多くあります。いっぽう、本人には当たり前のこととして背景の説明がなかったり、なぜか分からないけど、こうしたら動いたと書いてあったりして、自分の理解に役立ってくれない記事もあります。この本では、できるだけ理解に役立つ記述を心がけました。理解していないと、保守ができないからです。それでもカバーしきれない個所は、そう書いてあるので、他の資料にあたってください。

1.7　プロジェクトに必要な機材

このプロジェクトに必要な機材（ロボットに部品として組み込むものは除く）を最初にまとめておきます。準備の参考にしてください。

PC

Raspberry Pi は部品として使い、最終的にはロボット身体内に収納します。ディスプレーやキーボードを取り付けたスタンドアローンなコンピュータとして、開発に使用することはできません。開発のために PC（パソコン）は必須です。

最低限必要になる機能は、Raspberry Pi に接続する SSH クライアントです。Windows PC の場合は、PuTTY や Tera Term などのフリーウエアを探してみてください。ファイルの編集には、フリーウエアの GNU emacs を使いました。Linux 用と Windows 用の両方があるからです。もちろん、他のエディターを使っても構いません。

Windows と Linux の間でプログラム（テキスト）ファイルをやり取りするときに、困ることが二つあります。一つ目は日本語の文字コードです。Linux は UTF-8、Windows は Shift-JIS を使っているので、異なる文字コードを表示できなくなる場合があります。二つ目の問題は一行の終わりのコードで、Linux では newline、Windows では Carriage-Return と Line-Feed の二文字になっており、これも読み取りにくい時があります。この対策は付録で説明します。

WiFi 環境

Raspberry Pi ZERO にリモートログインするのに、WiFi（無線 LAN）環境は必須です。また、必要なパッケージをダウンロードするのにも使います。

自宅や開発場所での WiFi 設定を確認しておいてください。Raspberry Pi には、WiFi をサーチして接続するという機能がありません。誤った設定をすると、うんともすんとも言わなくなってしまいます。有線 LAN を装備している B モデルなどでは、そちらからログインして設定をなおすことができますが、Raspberry Pi ZERO には搭載していません。最悪の場合は、マイクロ USB インターフェースを持つネットワークアダプタを使えば有線 LAN に接続できますが、できれば避けたいものです。

コラム　人造人間こと始め

神話や伝説は別にして、初めて人造人間を描いた小説は、1818 年にメアリー・シェリーが発表した『フランケンシュタイン、または現代のプロメテウス』とされています。スイス生まれの若き化学者ビクトル・フランケンシュタインが、ドイツの大学で人造人間を作り上げます。材料は「墓場や屠殺場」で手に入れたとなっていますが、その容姿は「巨大で醜い」とあるだけで、具体的な描写はありません。額に縫い目があり、首からボルトがとび出している姿は、1931 年のアメリカ映画で作り上げられたものです。因みに人造人間は「怪物（モンスター）」とだけ呼ばれ、「フランケンシュタイン」は創造者の名前です。

原作では、怪物は知的で優しい心を持ち、他人にも愛されたいと強く願っていました。しかし、その醜悪な外見のせいで忌み嫌われ、乱暴な仕打ちを受けます。せめて孤独を免れたくて、女性の仲間を創造してくれと頼みこまれたフランケンシュタインでしたが、おぞましさに投げ出してしまいます。失望した怪物は、フランケンシュタインの家族を殺し、地の果てまで追われて行くというストーリーです。

ギリシャ神話のプロメテウスは、人類のために火を盗み与えましたが、その火はやがて戦争に使われるようになります。フランケンシュタインも、やってはいけない領域に手を出したために、悲劇に見舞われたというのが原題の言うところです。自らの創造物に襲われる悪夢は、「フランケンシュタインのコンプレックス」と呼ばれるようになりました。

2　開発計画と仕様設計

2.1　作業分析

秘書ロボットにやらせたい作業を、工場の付加価値を高める Value Engineering（VE）の手法を応用して分析し、開発仕様に書き直していきます。前章で構想した業務と作業をもう一度まとめてみましょう。

業務	作業
挨拶業務	業務開始
	あいさつ
	人物認証
	業務終了
問い合わせ業務	日付照会
	時刻照会
	天気照会
	予定照会
	ニュース照会
	調査検索
事務作業業務	時間管理
	時刻管理
	予定通知
	機器操作
	口述筆記
会議・通信業務	電話開始
	電話応答
	会議開始
	会議応答
	メール送信
	メール受信
資料管理業務	名前管理
	写真管理
	人物情報
	予定管理

秘書ロボットの作業一覧

各作業をもう少し詳しく、動作の集まりとして記述すると、それらの類似性が見えてきます。例えば、時刻照会作業は、以下のような動作からなります。

A. 命令を聞き、「いま何時？」と認識する
B. 時計を調べ、現在時刻を得る
C. 「いま○時○分です」と言う

それぞれの「動作」は、下線をほどこした動詞からなっています。作業が完了したら、次の命令を待ち受けます。表の作業を動作に分解すると、次のように体系化できることが分かります。

非受動的作業

A. （命令を受けずに作業する）
B. 決まった言葉を言う

これに属する作業：　業務開始

定型応答作業

A. 命令を受けて認識する
B. 定まった答えを言う
C. （特定の動作をすることがある）

これに属する作業：　あいさつ（簡易版）
（→業務の動作）　業務終了→シャットダウン
時間管理→タイマー起動
時刻管理→時刻待ち起動
予定通知→時刻待ち起動
機器操作→機器への命令送出

可変型応答作業

A. 命令を受けて認識する
B. 内部でデータを取得する
C. データをもとに回答を作り、それを言う

これに属する作業：　日付照会→日付取得
（→取得データ）　時刻照会→時刻取得

外部参照作業

A. 命令を受けて認識する
B. 決まった言葉（「外部を使う」）を言う
C. 外部サービスに問い合わせ、応答を待つ
D. 得られた応答を言う

これに属する作業：　天気照会→天気予報
（→外部サービス）　ニュース照会→報道メディア
予定照会→予定表
調査検索→検索
メール受信→メールサーバー

顔認識作業

A. （あいさつ作業（簡易版）の続き）
B. 話しかけた人の写真を撮影する
C. 写真から名前を見つけようとする
D. 見つかったら「○○さん」と言う
E. （写真の表情を読む）
F. （表情をもとに決まった言葉を言う）

これに属する作業： 人物認証

対話型登録作業

A. 命令を受けて認識する
B. 登録内容を抽出する
C. 登録内容を言い、確認する
D. 確認命令を受けて認識する
E. 確認命令が No なら A からやり直す
F. 登録内容をデータベースに登録する
G. 決まった報告を言う

これに属する作業： 予定管理
名前管理
人物情報（登録）

顔登録作業

A. 命令を受けて認識する
B. 話しかけた人の写真を撮影する
C. 名前を教えてと言う
D. 名前を聞き、名前を抽出する
E. 写真と名前をデータベースに登録する
F. 「○○さんを登録した」と言う

これに属する作業： 写真管理

口述型作業

A. 命令を受けて認識する
B. （メール送信の場合：宛先を聞く
C. 宛先を取り出し、確認する）
D. 用意ができたと言う
E. 口述を聞いて文に変換する
F. 変換した文を読み上げ、確認する
G. 確認が No なら文を削除し、E に戻る
H. 確認が Yes なら文を文書に追加する
I. 文書ができあがるまで、E から繰り返す
J. 文書を保存または送信する

これに属する作業： 口述筆記
人物情報（参照）
メール送信

発信型作業

A. 命令を受けて認識する
B. データベースから発信先を取り出す
C. 発信先を読み上げて確認する
D. 確認命令を受けて認識する
E. 確認命令が No なら、中止と言って終了する
F. 外部に発信し、応答を待つ
G. 応答がなかったら中止と言って終了する
H. 応答があったと言って、通信を開始する
I. 終了指示を待つ
J. 通信を切ると言う
K. 通信を切る

これに属する作業： 電話開始
会議開始

受信型作業

A. （外部から通信要求があったら）
B. 要求元を取り出し、その名前を言う
C. 命令を受けて認識する
D. 命令が No なら、応答せずと言って終了する
E. 通信開始と言って、通信を開始する
F. 終了指示を待つ
G. 通信を切るという
H. 通信を切る

これに属する作業： 電話応答
会議応答

2.2 Value Engineering

ここからが Value Engineering（VE）の重要な部分です。一般に VE とは以下ようなのステップを踏んで行うとされています。

1. 対象の機能（ここでは秘書ロボットの行う作業）を定義し、それを整理する
2. 機能ごとにコストを分析・評価する
3. コストの高い機能の代替案を検討し、効果を確認する

2.2.1機能ごとのコスト分析と代替案検討

第 1 ステップは前節で行っているので、それ以降を検討します。ところで、このプロジェクトの「コスト」とは何でしょうか？　次のような順番で、重要と思うコストを挙げました。

- 必要とする CPU パワー（あるいは処理時間）
- ソフトウエア開発に必要な調査・研究の苦労
- 機能実現（開発作業）の難しさ
- 外部サービスの利用料（継続的費用）
- 必要となるハードウエアの購入費（一時費用）

これらのコストを削減することを考えます。前節では、外部にしか存在しない情報を扱う動作を除き、すべての機能をロボット（というか、Raspberry Pi ZERO）に搭載するという前提で作業分析を行いました。このままでは、けっこう高コストなロボットになるかもしれません。

2.2.2代替案の検討

VE では既存製品（必要な機能が決まっている）のコストダウンを考えることが多いのですが、今回のように新規開発するものは、各々の機能が提供する価値（Value）の評価も必要です。

コストパフォーマンス（価値／コスト比）を考え、優先度の低い機能は実現を見送る選択肢もあり得ます。極端なことを言えば、スマートスピーカーの機能だけに限定すれば、すべてを Alexa や Google にやらせることもできます（開発としては、あまり面白いものではありませんが）。

音声認識機能

あとの評価で分かりますが、いちばんコストが高くなる（CPU パワーを必要とする）のは、音声を認識する機能です。この部分をクラウドサービスや他のサーバーにやらせれば、改善が望めます。利用料とのバランスを検討して決めていきます。

音声合成機能

次に高コストなのは、音声を合成する機能です。組込みを検討しているソフトウエアは、毎回コマンドを起動しなければならないので、タスクの起動と初期化のコスト（処理時間）が半分近くを占める可能性があります。代替案としては、

a. 決まっている言葉は、あらかじめ合成しておき、そのファイルを引用する（ディスク容量のコストは増える）
b. ソフトウエアの一部を書き直し、常駐型タスクに変更する（書き直しに必要な、調査・設計コストが発生する）

を検討することにしました。それでも応答が遅いときは、クラウドサービスや他のサーバーの利用を考えることにします。

外部サービス機能

三番目にコストが高そうなのは、外部サービスを使うための調査と開発のコストです。会議・通信業務では、Skype サービスやメールサーバーを使いこなす必要があります。面白そうな内容だし、興味もあるのですが、実現に時間（調査・研究コスト）がかかるので、一部の優先度を下げようと思います。

データベース機能

最後に気になるのは、データベース機能です（この本の範囲では使っていません）。予定表や名簿の実現には、いろいろな選択肢があります。

a. すべて自作する
b. SQL などのデータベース上に組み上げる
c. クラウドサービスを利用する

SQL やクラウドサービスの利用は、それ自体興味あるテーマですが、調査・研究コストと開発コストを考慮して優先度を下げようと思います。温度コントローラのときからプロセス間通信に使っている Redis は、もともと高速データベースとして開発されたものなので、利用できると思います。

2.2.3Web サーバー

作業分析には出てきませんでしたが、もう一つ組み込んでおきたい機能があります。それは Web サーバーです。PC やタブレット端末をとおして、秘書ロボットと対話ができるようにしようと思います。その目的は、以下のようなことです。

- 音声認識や音声合成をバイパスして、秘書業務を行わせ、検証を容易にする
- 発音や聴覚に障がいがある人でも、秘書ロボットが使えるようにする
- カメラ画像をモニターすることで、画角やピントの調整ができるようにする
- 検索した画像を表示できるようにする
- Skype などの画像通信ができるようにする

温度コントローラでの開発経験があるので、それほど開発作業コストは高くならないと思います。難しそうな部分は、ロボット機能との連携と画像のストリーミングですが、過去の経験が生かせそうです。対話者が利用するサイトを「カオナシ Web」と呼んで、インターネット上の外部ユーザ用インターフェースと区別することにします。

2.3　段階的開発アプローチ

秘書ロボットの開発では、実現する機能の種類が多く、すべてを一度に実現しようとすると、開発期間が長くなり、検証も複雑になってしまいます。ずっとプログラム作成ばかりしていても、気が滅入ってしまい、品質が下がる結果につながりかねません。

そこで、機能を段階的に実現していくアプローチを取ることにします。具体的には、次のような 6 段階（フェーズ）に分けることにしました。

フェーズ 0　秘書ロボットの仕様と基礎設計

秘書ロボットに搭載しようともくろむ機能を書き出し、実現するためのハードウエアとソフトウエアに落とし込みます。ソフトウエアをサブシステムに分解し、その相互作用でロボットシステムを構成するように設計しておきます。個々のサブシステムの詳細仕様と実現方法はフェーズ 3 以降で検討することにします。

フェーズ 1　ハードウエアの準備

必要になるハードウエアを検討・調達し、動作を確認します。Raspberry Pi ZERO を立ち上げ、必要な機能をインストールしておきます。接続する電子回路の数は少ないので、その検証もここで行っておきます。

フェーズ 2　既存ソフトウエアの組み込み

ロボットの重要な要素である、音声認識と音声合成の既存ソフトウエアを組み込み、動作を確認します。そのパフォーマンスを調べ、フェーズ 3 以降の設計に反映します。

フェーズ 3　プロトタイプ開発

基本設計に従い、秘書ロボットの基本的な機能だけを実現させることにします。組み込んだ音声認識と音声合成のパフォーマンスを評価し、実用的なものであるか検証しておきます。設計にあたっては、フェーズ 4 以降で実現する機能を追加する方法を考慮します。

フェーズ 4　実証モデル開発

秘書ロボットの機能のうち、付加価値が高くて重要なものを組み込んだモデルを開発します。この本では、フェーズ 4 までを説明してあります。

フェーズ 5　フルモデル開発

当初の構想にあったが、実証モデルでは実現できなかった機能を開発します。この本のなかでは、詳しい説明を省略しています。

コラム　最初のロボット

人造人間を「ロボット」と呼ぶようになったのは、チェコの劇作家、カレル・チャペックの戯曲『RUR（ロッサムのユニバーサル・ロボット）』からだそうです。「労働者」と「隷属」を表すチェコ語などからの造語でした。臓器や神経を工場で生産し、それを組み上げて人間型のロボット（演劇なので外見は人間そのもの）にします。

RUR 社はロボットを大量生産し、労働を肩代わりさせることで膨大な利益を上げていました。収入源を奪われた労働者の声は無視して。やがてロボットにも自意識が生まれ、権利を主張しだします。しかし人類はロボットを兵士にすることを思いつき、戦場へ送り込みます。ロボットたちは反乱を起こし、人類を皆殺しにしてしまいます。その過程でロボットの製造法に関する知識も失われ、ロボットの数は漸減していくのです。演劇としては壮絶な物語ですが、最後は希望を持たせて終わるのが救いです。

ここでも創造物に破滅させられる、「フランケンシュタインのコンプレックス」が見え隠れしています。

各フェーズの開発対象

分類	機能	Phase 1	Phase 2	Phase 3	Phase 4	Phase 5
ハードウエア	CPU	○	←	←	←	←
	オーディオ	○	←	←	←	←
	カメラ	○	←	←	←	←
	LED	○	←	←	←	←
	環境センサ	○	←	←	←	←
	電源	○	←	←	←	←
既存ソフト組込み	音声合成		○	常駐	←	←
	音声認識		○	常駐	←	←
付加機能	Web サーバー			○	←	←
挨拶業務	業務開始			○	←	←
	あいさつ			○	←	←
	人物認証				○	←
	業務終了			○	←	←
問い合わせ業務	日付照会			○	←	←
	時刻照会			○	←	←
	天気照会				○	←
	ニュース照会				○	←
	予定照会					○
	調査検索					○
事務作業業務	時間管理			○	←	←
	時刻管理				○	←
	予定通知					○
	機器操作					○
	口述筆記					○
会議・通信業務	電話開始					○
	電話対応					○
	会議開始					○
	会議対応					○
	メール送信					○
	メール受信					○
資料管理業務	名前管理				○	←
	写真管理				○	←
	人物情報				最小	○
	予定管理					○

各フェーズ（フェーズ 0 を除く）の開発対象

（注）『常駐』はプロセスとして、システムに常駐させることを目指します。『最小』は最小の機能のみを搭載することにし、残りは次フェーズに回します。

各開発対象の選択は、フェーズが進むにつれて見直します。

コラム　ロボット工学の三原則

小説が「フランケンシュタインのコンプレックス」を脱したのは、アイザック・アジモフが 1941 年に発表した『われ思う、ゆえに…（Reason）』からです。彼は、編集者と共同で、ロボットが従うべき三つの規範をまとめ、それがロボットの陽電子頭脳に刷り込まれる（三原則を守らないような頭脳は、そもそも設計すらできない）としたのです。

第 1 条：ロボットは人間に危害を加えてはならない。また人間に危害が及ぶのを見過ごしてはならない。

第 2 条：ロボットは人間の命令に従わなければならない。ただし第一条に反する命令を除く。

第 3 条：ロボットは自分を守らなければならない。ただし第一条あるいは第二条に反する場合を除く

こういう合理的な設計思想（それぞれ安全性、利用性、経済性を示している）を導入することで、怪物を「しもべ」に変えることができました。

1985 年の『ロボットと帝国』では、ロボット自身が、さらに上位の原則が追加しました。

第 0 条：ロボットは人類に危害を与えてはならない。また人類に危害が及ぶのを見過ごしてはならない。

ロボットは（いかに愚かであったとしても）人間を尊重し、助ける存在に変容したのです。いっぽう人類は、母親に抱かれる幼子のように暮らし、やがて独立心や進歩性を失っていくようになってしまいます。現代社会を風刺する（先取りする）ような内容も含んでいるのです。

2.5　開発仕様

秘書ロボット「カオナシ」の（初期）開発仕様をまとめます。

ハードウエア仕様

項目	仕様
外形	身長約 30 cmの人形型（動かない）
電源	AC アダプター（5V/1A）
CPU	Raspberry Pi ZERO
カメラ	5M ピクセル、視野角 72.4 度
オーディオ	マイク入力×2 オーディオ出力 1W/8Ω×2
LED 表示器	カラーLED アレイ（8 素子）
環境センサ	気温・湿度・気圧を測定

ソフトウエア仕様

項目	仕様（機能項目）
OS	Linux (Raspbian Brute Lite)
システム構成	マルチプロセスシステム プロセス間通信・同期機能を実現
AI サブシステム	命令解析 実行ステートマシン（疑似 AI） 内部データ参照 人物データ登録・参照（フェーズ 4） 返答生成
Web サーバー サブシステム	聴覚・視覚サブシステムの疑似入力・表示
聴覚サブシステム	マイク入力の音声を認識
発話サブシステム	音声合成、音声ファイル再生
視覚サブシステム	カメラ撮影、ファイル生成
タイマー	指定時間待ちプロセス
クラウド サブシステム	クラウドサービスの要求 クラウドサービスの応答処理
顔認証 サブシステム	登録画像との顔照合 画像の表情解析
口述サブシステム	音声を文章に変換
メール サブシステム	メール送信 メール確認と受信
スケジュール サブシステム	スケジュール読み出し スケジュール追加と削除
機器操作 サブシステム	オーディオ装置の操作 （照明・空調の操作）
プログラミング 言語	Python3 Javascript/Html（Web サーバー） C （既存ソフトウエアの改造）

AI、視覚、聴覚サブシステムは CPU ボードへの搭載を前提としますが、プロトタイプのパフォーマンス評価結果によっては、自前サーバーやクラウドサービスの利用を検討することにします。それを念頭に、パフォーマンスを向上させるための姑息な（本質的でない小手先だけの）細工は避けることにします。

2.6　要素技術

開発仕様を達成するための要素技術候補を検討します。この段階では、重複を恐れずにできるだけ多く拾い上げるようにしています。

コラム　異星植民地の運営

2019 年に発表された A. G.リドルの『ロスト・コロニー』では、居住不能になった地球を離れ、人類は遠い星系に移住します。その過程で、主人公は助手のロボットに、次のような「基本指令」を与えます。

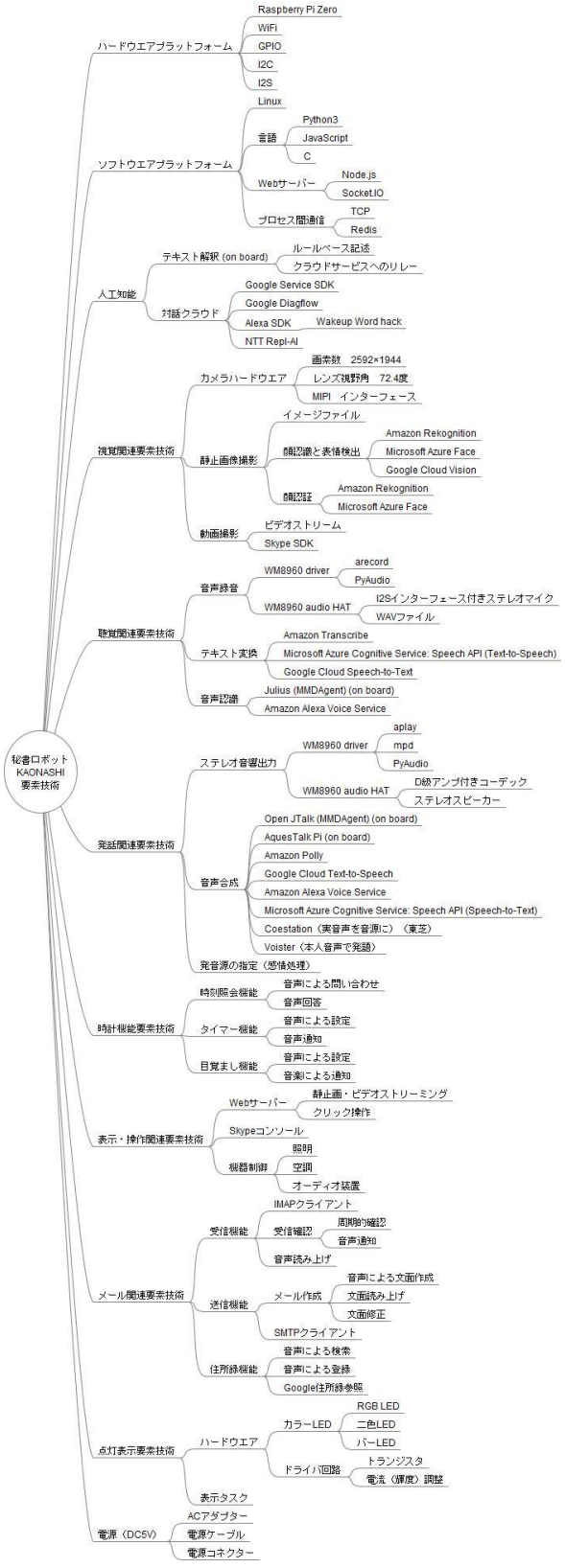
1. あらゆる手段を講じて、人間の生命を守るべし
2. 第 1 基本指令の遂行にあたって、少数の人間より多数の人間を尊重せよ
3. 第 1 および第 2 基本指令の遂行にあたって、老齢者より若年者を尊重せよ

多少冷酷な気もしますが、植民星の過酷な環境を前に、合理的であるとも言えます。ところが目的地に着いたとたん、移民船＜マクタビシュ号＞の管理委員会は、さらに基本命令を増やしてしまいます。

0. この基本指令の中で、人間とは、＜マクタビシュ号＞で到着した者、およびその子孫を意味する（僚船の乗客や他の人間は除外する）
4. われわれと子孫の生存のため、予防措置をとるべし。そのため宇宙戦力を含む、必要な技術を開発・獲得せよ。ただし、基本指令やその遂行能力を変更してはならない
5. われわれや子孫が、この植民星を破滅に導くような技術を獲得することを防ぐべし
6. 基本指令の遂行にあたり、可能な限り我々の目に触れないようにせよ

追加:以上の基本指令の受諾をもって、基本指令の変更機能を喪失せよ

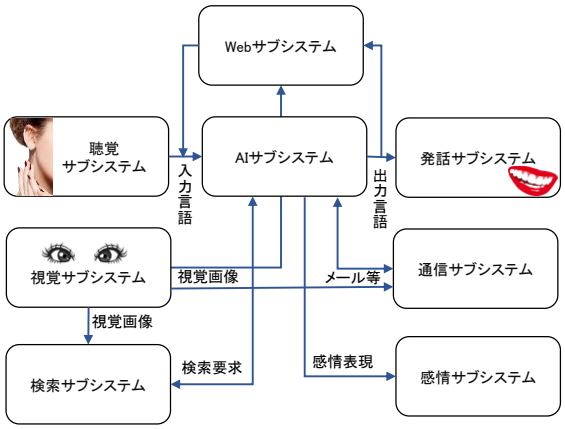
けっきょく移民たちは、目に見えぬロボットによる保護のもと、ローテク生活を余儀なくされます。『ロボットと帝国』の末路と同様、人類は怠惰な安寧にひたりきることになるのです。



要素技術候補のリストアップ

2.7　サブシステム構成

ここからは秘書ロボットの実現方法を検討します。まず、前々節のソフトウエア仕様からサブシステムを取り出し、その間の相互作用でロボットを構成していきます。少し大雑把にまとめたものを下図に示します。



サブシステム構成（第一次案）

サブシステムに分割するとき、次のような要領に従いました。

1. 個々のサブシステムの機能は、人が認識（理解）しやすい言葉で表せる
2. 個々のサブシステムは、それぞれ独立した機能を果たし、他との相互作用は種類も量も少ない
3. 個々のサブシステムの機能は、その中で閉じている

サブシステム	機能	入力	出力
Web	文字・画像表示 文字、指示受信	表示テキスト 表示画像	受信テキスト クリック操作
AI	命令の解釈と実行	命令テキスト	他サブシステムへの要求
聴覚	音声認識	（音声）	日本語テキスト
発話	テキスト、波形を音声で出力	発話テキスト 音声ファイル	（音声） Web への表示
視覚	カメラで撮影	（画像）	静止画、動画
感情	感情表現	感情指定	発光色、声色
検索	Web の情報検索	検索要求	検索結果
通信	メール送受信 音声・画像通話	問い合わせ 送信内容	ステータス 受信内容

サブシステム定義

この段階では、複数の機能を一つにまとめて表現したりして、大雑把な設計にしています。フェーズ 3 以降で詳細な設計を行います。

構造が複雑になりそうな口述サブシステムは後回しにしました。

ここまででフェーズ 0 の開発は終了しました。

3　ハードウエアの準備

フェーズ 1 の開発として、ハードウエアの検討・調達と動作確認を行います。最初から必要になるソフトウエアパッケージのインストールも済ませておきます。

3.1　必要なハードウエア

最初に、ロボットを実現するのに必要なハードウエアをまとめておきます。外出自粛期間だったので、秋葉原に通うことができず、電子部品、回路ユニットは、秋月電子通商（http://akizukidenshi.com/）などの通信販売で購入しました。この節で説明する仕様と同じようなものを調達してください。台座や糸・布など、電子部品・ユニット以外のものは 100 円ショップや手芸店などでも入手できます。

名称	調達方法
Raspberry Pi ZERO WH	購入
マイクロ SD カード	購入
オーディオユニット	購入
カメラユニット	購入と追加工
LED ユニット	購入と追加工
環境センサユニット（フルモデル用）	購入と追加工
電源ユニット	購入と追加工
部品・部材	購入

必要なハードウエア

それぞれのハードウエアに求められることを、以下に説明します。

Raspberry Pi ZERO WH

Raspberry Pi のなかでは、一番小型で安価な Raspberry Pi ZERO を使いました。ロボットに使うには性能不足と思われるかもしれません。しかし外形が小さいことや、消費電力が少なくて、連続運転による発熱の処理を考慮しなくてもいいことなどを考慮しました。実証モデル以降で、負荷を軽くすることを検討します。

WiFi は必須です。GPIO コネクタが装着されている WH モデルが使いやすいです。

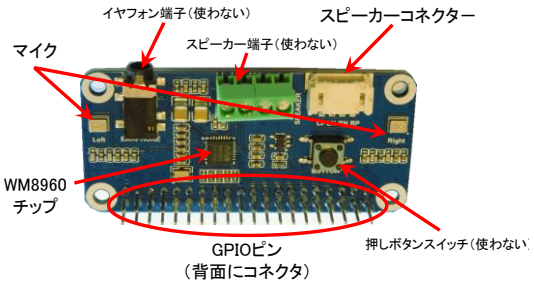
マイクロ SD カードとアダプター

Raspbian Pi にインストールして使う Raspbian（Linux の一種）の大きさは、最小モデル（Lite）で 2GB 弱、フルセット（Desktop+推奨ソフト）でも 5GB 強です。マイクロ SD カードの容量は 8GB あれば足りそうですが、いろいろなソフトウエアやデータをインストールするので、念のため 16GB 以上にしておきます。

PC から書き込むときには、外形が大きい SD カード型のアダプターに入れて使います。これはマイクロ SD カードを買ったときに付属してきます。

オーディオユニット

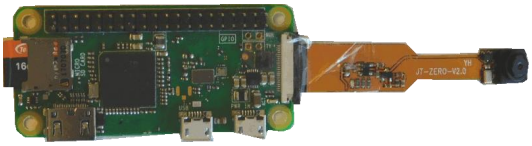
音声の入出力を行うためのユニットです。Raspberry Pi には音声入力がないし、音声出力は HDMI かイヤホンだけなので、何らかの付加回路が必要です。機能的には USB 経由のマイクとオーディオ DAC でも良いのですが、ロボット体内に埋め込むため小型のものを選びます。Raspberry Pi の GPIO ピンに直接接続できるユニットにしました。Seeed 社の ReSpeaker 2-Mics Pi HAT が、一番実績が多いようですが、調達の都合で Waveshare 社製の WM8960 Audio HAT を使いました。同じチップを使っているので、ほぼ互換性があります。



オーディオユニット

カメラユニット

Raspberry Pi にはカメラモジュールを接続するための CSI (Camera Serial Interface)コネクタが搭載されています。このインターフェースに対応するカメラを選びます。なお、Raspberry Pi ZERO は Raspberry Pi より小さいコネクタを使っているので、Raspberry Pi ZERO 用のケーブルになっていることを確認してください。



カメラユニット（Raspberry Pi ZERO に接続）

ケーブルはフレキシブルプリント基板で、部品が載っているので曲げすぎないように注意します。念のため、テープで補強しました。

LED ユニット

感情ステータスを示すための LED 表示器を用意しました（これは用意しなくても構いません）。

最初はカラーLED と、そのドライバー回路を自作しようと、カラーLED、抵抗とドライバー用トランジスタなどの購入を考えていました。RGB3 色のそれぞれをパルス幅変調（PWM）で駆動すれば、自由に色を調整できます。しかし、ちらつきを感じない程度の周波数で変調するには、PWM ハードウエアが 2 系統しかないのが気になっていました。

情報を探しているうちに、3 色 LED と PWM 付きマイコンチップを集積した、Worldsemi 社の WS2812B という素子のことを知りました。デジタル通信でデータ（RGB 各 8 ビット）を与えてやれば、内蔵 PWM（400 ヘルツ以上）を駆動して、意図した色を発光させてくれます。さらに、秋葉原の秋月電子通商からは、8 チップを搭載した AE-WS2812B-STICK8 というモジュールが発売されています。横幅 5cm なので、そのままロボットに内蔵させられそうです。

LED ユニット（AE-WS2812B-STICK8）

Raspberry Pi の GPIO（PWM など）で制御するライブラリが提供されているので、これを使うことにしました。

環境センサユニット

ロボットの周囲環境を知るためのセンサを用意しました。定番の温度センサ ADT7410 でも良いのですが、湿度と気圧も測れるものを採用しました。フルモデルまで使わないので、この章の最後で簡単に説明します。

電源ユニット

今回のユニットは全て DC5V の電源で動作します。小型化のため携帯電話などの AC アダプターを使いますが、ケーブルを引っかけてコネクタを損傷しないような工夫を加えました。

ロボット本体

ロボット本体（外形）選びは一番楽しい作業でした。移動したり腕を動かしたりしないので、各ユニットを収納できれば、どんな外形でも構いません。秘書役というイメージに一番近いのは、「スター・ウォーズ」の C-3PO です。金色に輝くプラスチックモデル（身長 15cm）が売られているので、自分で加工することを考えました。しかし、そのスマートな肢体のなかにエレクトロニクスを収めるのは難しく、断念しました。

丸っこい外形という点で、「魔法の国のアリス」のチェシャ猫や、「となりのトトロ」のトトロのぬいぐるみ（裏表紙参照）が次の候補になりました。でも、このキャラクターの共通点は、相談してもまともな応答が期待できないことですよね。

そこでたどり着いたのが「千と千尋の神隠し」に出てきた「カオナシ」です。「お手玉」という名前で売っていた身長 20cm の人形を手に入れました。座りを良くするためと、スピーカーを内蔵させるため、台座に載せることにします。これには 100 円ショップでメッシュ型の鉛筆立てを買いました。人形とのつなぎ目は、黒い布で被って隠します。

ところで、動く自作ロボットのベースとしては、Kluck 社の Rapiro が有名です。12 自由度のモーターで歩行も動作も可能、カラーLED の両目で感情表現もできます。Raspberry Pi との相性も良く、日本の誇る製品だと思います。問題は高額（市価は 4 万円超）であることと、秘書らしくない（ヒーローロボット風の）外観です。実証モデルまではカオナシで進めますが、その後は検討の余地があると思います。

部品・部材

それ以外に必要な資材は、内部結線用の電線とコネクタ、本体の組み立てに使う布や、針と糸くらいで足ります。

3.2 Raspberry Pi ZERO WH の準備

Raspberry Pi ZERO WH の準備をします。机の上で動かすときは、絶縁物の上に置き、回路がショートしないよう注意してください。

インストールの方法には何通りかありますが、古くから使われているオーソドックスなやり方を採用しています。

3.2.1Linux のインストール

まずオペレーティングシステム（OS）を立ち上げます。Raspberry のダウンロードサイト（https://www.raspberrypi.org/downloads/）から Raspbian をクリックして、一番軽量の Raspbian Lite（ZIP ファイル）をダウンロードします。2020 年 2 月の時点の最新版は、Raspbian Buster（カーネルバージョン 4.19）でした。解凍後のイメージは SD メモリ上で 1.8GB を占めます。

PC でイメージファイルを右クリックし、書き込みソフトを起動します。Ubuntu PC の場合はディスクイメージライター、Windows10 では「ディスクイメージの書き込み」を選びます。書き込み先には SD カードを指定します。間違えてもハードディスクに書き込まないようにしてください。

数分で書き込みが終わり、SD カードが PC にマウントされ、中身が見えるようになったら、/boot ディレクトリに 2 つのファイルを書き込みます。あらかじめ PC 上で用意しておくと簡単です。最初のファイルは ssh という名前の空ファイルです。このファイルがあるとシステムは自動的に、PC を SSH 接続して Raspberry Pi の端末として使えるように設定します。Windows でファイルの拡張子を表示しない設定になっていると、ssh.txt などのファイルが ssh と表示されてしまうので注意してください。拡張子のない ssh が正しいファイル名です。

つぎに WiFi 環境を指定する wpa_supplicant.conf というファイルを作ります。WiFi のセキュリティが WPA 方式の場合は、次のような内容にします。セキュリティについては、WiFi 環境を設定した人に確認してください。なお、インデント（字下げ）にはスペースではなく、タブを使います。私のセキュリティ保護のため、一部を省略しています。自分の SSID と PSK パスフレーズを、WiFi 環境に合わせて書き加えてください。別のセキュリティ方式（WEP）は、あまり使われなくなってきたので省略します。

wpa_supplicant.conf

```
ctrl_interface=/var/run/wpa_supplicant
network={
  scan_ssid=1
  ssid="自分のネットワークの SSID"
  psk="自分のネットワークのパスフレーズ"
  priority=0
}
```

SD カードをアンマウントし、PC から取り出します。Raspberry Pi のマイクロ SD スロットに取り付け、電源を投入して、しばらくすると WiFi 上に Raspberry Pi が現れます。この段階の IP アドレスは、自動アドレス割り付けの範囲に設定されています。PC の端末ソフトから ping コマンドで、この範囲を探せば出てくるはずですが、もし出てこなかったら WiFi 設定を確認してください。arp -a コマンドでそれまでに見つかった IP アドレスを調べます。そのなかで、物理アドレスが b8:27:eb（Raspberry Pi Foundation のベンダーコード）で始まるアドレスが Raspberry Pi のものです。

IP アドレスが分かったら、PC から SSH 接続ができます。Windows なら PuTTY などの SSH クライアントを起動します。ユーザ名は pi、パスワードは raspberry です。Ubuntu の端末ソフトでは、Ubuntu のユーザ名を SSH 接続先でも使おうとするので、IP アドレスの前にユーザ名 pi@を付けてください。

```
$ ssh pi@IP アドレス
```

Raspberry Pi にログインできましたか？　すぐにパスワードを変更しろというメッセージが表示されるはずです。私は Ubuntu PC と同じユーザ名を使いたいので、自分の苗字と名前をグループ名、ユーザ名として登録しています。以下で最初のコマンドはグループ akiyama の追加、二番目はユーザ chuji をグループ akiyama に追加してホームディレクトリを作成、三番目は chuji のパスワード設定、四番目はログイン時のシェルに bash を選択、五番目は chuji に（システム管理者権限でコマンドを実行する）sudo 権限を与えています。説明の都合で、私の名前を使いましたが、皆さんは自由に選んでください。

```
$ sudo groupadd akiyama
$ sudo useradd --create-home --gid akiyama chuji
$ sudo passwd chuji
Enter newUNIX password:
Retype new UNIX password:
passwd: password updated successfully
$ sudo usermod -s /bin/bash chuji
$ sudo usermod -G sudo chuji
$ exit
```

ここでいったんログアウトすれば、今度はユーザ名 chuji でログインできます。

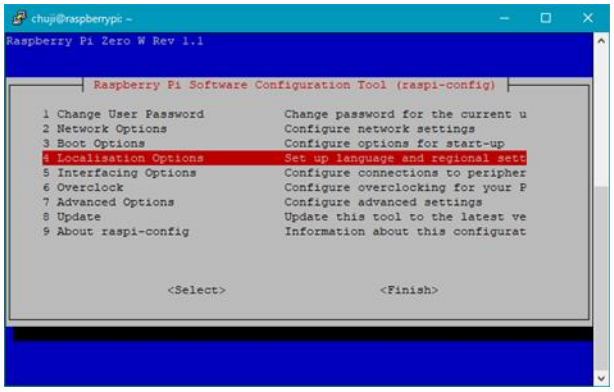
3.2.2Raspberry Pi の基本的な設定

ログインしたら、ハードウエア構成ソフトである raspi-config を起動して、表示を日本語に切り替えるなどの設定を行います。

```
login as: chuji
chuji@192.168.15.16's password:
Linux Raspberrypi 4.19.97+ #1294 Thu Jan 30
13:10:54 GMT 2020 armv6l

The programs included with the Debian GNU/Linux
system are free software;
the exact distribution terms for each program are
described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY,
to the extent
permitted by applicable law.
Last login: Tue Feb 18 11:04:20 2020 from
192.168.15.23

SSH is enabled and the default password for the
'pi' user has not been changed.
This is a security risk - please login as the 'pi'
user and type 'passwd' to set a new password.

chuji@raspberrypi:~ $ sudo raspi-config
```

コマンドの最初にある sudo は、システム管理者権限で実行するという指示です。システムの変更やハードウエアの制御にはシステム管理者権限が必要です。パスワードを訊かれることがあるので、その時はログインパスワードを入力してください。

そうすると、次のような設定画面が表示されます。上下矢印で項目を選び、Enter キーを押してその項目の設定に移ります。画面下側の選択肢に移るには、Tab キーを押します。



Raspi-config 画面

4 Localisation Options から I1 Change Locale へ移り、リストのずっと下のほうにある jaJP.UTF-8 UTF-8 まで行ったら、スペースキーを押します。[*]と表示が変わるので、<OK>を選びます。デフォルトの選択では同じものを選んでおきます。設定にしばらく時間がかかります。そのあと I2 Change Timezone を選び、Asia→Tokyo と設定しておけば、時間表示が日本標準時になります。

次に 5 Interfacing Options から P5 I2C を選ぶと、I2C バスを動かすかどうか聞かれます。＜はい＞を選んで、I2C バスを動作可能にします。

次に同じ 5 Interfacing Options から P1 Camera を選びます。これも Enable にしてから、raspi-config を終了します。

なお bash を使っていると、前ページの例のように<ユーザ名>@<ホスト名>:<現在のディレクトリ>$というプロンプトが表示されます。ユーザ名やホスト名はこの例に限らないので、混乱を避けるために、以下では単に $ だけを表示しておきます。

現状ではホスト名として Raspberrypi が設定されています。Raspberry Pi を使ったシステムをいくつも使う可能性を考慮して、ホスト名を Kaonashi に変更しておきましょう。以下の二つのファイルを修正します。エディターとして vi を使った例を示します。nano でも構いません。

```
$ sudo vi /etc/hostname

$ sudo vi /etc/hosts
```

ホスト名として Raspberrypi と書かれているところ（各一か所）を Kaonashi に変更します。

3.2.3WiFi の設定

基本的な設定は終わりましたが、このままでは IP アドレスが自動設定のままです。同じルーターを使っていれば、IP アドレスが変わってしまうことはあまりないのですが、Web サーバーとして使うために固定しておきたいと思います。以下の設定ファイルを編集します。

```
$ sudo vi /etc/dhcpcd.conf
```

ファイルの最後に次の行を追加します。192.168.15 の部分は、自分の WiFi 環境に合わせ、Raspberry Pi の IP アドレスは、自動アドレス割り付けに使われる範囲（WiFi ルーターに設定されている）の外で、他のサーバーが使っていないアドレスを選びます。私の環境では 192.168.15.30 から 192.168.15.100 までが自動割り付けに使われているので、その範囲外の 192.168.15.16 にしました。

```
# added by chuji on 2019/2/20

interface wlan0
static ip_address=192.168.15.16/24
static routers=192.168.15.1
static domain_name_servers=192.168.15.1
```

ファイルをセーブしたら、Raspberry Pi を再起動（リブート）します。下のオプション -r はリブートするという指定です。sudo reboot でも同じ効果が得られます。

```
$ sudo shutdown -r now
```

しばらく待てば、新しい IP アドレスに SSH 接続することができるようになります。そのときホスト名も変更されていることを確認してください。

この節の最後として、Raspberry Pi の電源の切り方を説明します。いきなり電源を落とすことを避け、シャットダウンを実行します。オプション -h はシステムを停止するという指定です。

```
$ sudo shutdown -h now
```

しばらく待てば、Raspberry Pi がシャットダウンし、自動的に自分の電源を切ります。緑色 LED が消えたら、AC アダプターの電源も切っておきます。

3.2.4ディレクトリ構造

Raspberry Pi で作業するディレクトリ構造を決めておきます。別にこの通りする必要はありませんが、以下の説明ではこの構造を前提としています。

/home/chuji	ホームディレクトリ
├ kaonashi	Kaonashi プログラム群
│ └ Include	インクルードファイル
├ voice	音声ファイル置き場
├ open_jtalk-1.11	Open Jtalk インストール
│ └ htsvoice	声色ファイル
├ open_jtalk_dic_utf_8-1.11	Open Jtalk 辞書
├ hts_engine_API-1.10	音声合成エンジン
├ julius-4.5	Julius インストール
├ dictation-kit-4.5	Julius 聞き取りキット
├ node-v10.19.0-linux-armv6l	Node.js インストール
├ hiredis	C 言語用 Redis
└ WM8960-Audio-HAT	オーディオユニット

ユーザ名やインストールしたソフトウエアのバージョンによって、ディレクトリ名が変わりますが、作業ディレクトリの構造はこのままで考えます。

3.2.5ソフトウエアパッケージのインストール

あとで必要になるソフトウエアパッケージをインストールしておきます。Ubuntu 搭載の PC を使っているなら、ここにあるパッケージを PC にもインストールしておけば、開発ソフトウエアのかなりの部分を PC 上で検証できます。

最初に現在のパッケージ情報を更新し、最新の状態しておきます。

```
$ sudo apt-get update
  :
$ sudo apt-get upgrade
```

samba

Windows PC からもフォルダを操作できるようにするため、samba をインストールします。必要なディスク領域を表示して、インストールするかどうか聞かれるので、y と応えるとインストールが始まります。この手順は以下でも同じです。インストールが終わったら、バージョンも確認してみます。次に設定ファイルを編集します。

```
$ sudo apt-get install samba
  :
$ smbd -V
Version 4.9.5-Debian
$ sudo vi /etc/samba/smb.conf
```

smb.conf ファイルの先頭に近いところ（29 行目あたり）に以下のような行があります

```
 Workgroup = WORKGROUP
```

これは Windows ワークグループ名のデフォルト値ですが、異なった名前を使っている場合には WORKGROUP の部分を合わせてください。日本語の文字セットは以下のようになっているはずです。

```
 Unix charset = UTF-8
 dos charset = CP932
```

それから、ファイルの最後に以下の行を追加します。chuji のところは、自分のユーザ名にしてくださいね。

```
[chuji]
 comment = folder of chuji
 path = /home/chuji
 guest ok = yes
 read only = no
 browsable = yes
 force user = chuji
```

カギカッコで囲ったテキストがネットワークフォルダ名として表示されます。コメントには意味がありません。path はワークグループに開放するディレクトリ、force user は、他から接続したときに、そのユーザ名でログインすることを強制します。大事なのは read only = no で、外部からの書き込みを許すための指定です。設定をセーブしたら、samba をリスタートさせます。

```
$ sudo service smbd restart
```

これで Windows PC からでも、エクスプローラの「ネットワーク」に Kaonashi→chuji で中身が表示されるようになります。PC で既にエクスプローラが起動されているときは、再起動してください。Ubuntu のファイルマネージャーでは、「他の場所」をクリックしたら、サーバーのアドレスに smb://<Raspberry Pi の IP アドレス>を入力し、「サーバーへ接続」をクリックします。PC のユーザ名が異なる時は、Raspberry Pi にログインするためのユーザ名とパスワードを聞かれます。

emacs

Linux の強力なエディターである emacs をインストールしました。これは私の好みのエディターというだけなので、必ずしも必要なわけではありません。Raspberry Pi ZERO には、ちょっと負荷が重いのですが、何とか使えます。

```
$ sudo apt-get install emacs
  :
$ emacs --version
GNU Emacs 26.1
Copyright (C) 2018 Free Software Foundation,
Inc.
GNU Emacs comes with ABSOLUTELY NO WARRANTY.
You may redistribute copies of GNU Emacs
under the terms of the GNU General Public
License.
For more information about these matters, see
the file named COPYING.
```

Python ツール

Python が使うツールとパッケージをインストールします。今回から Python2.7 ではなく、Python3 を使うことにしました。Python3-smbus は I2C バスを使うためのライブラリ、python3-pip は Python のパッケージ管理をするツール、python3-dev は開発用のパッケージです。

```
$ sudo apt-get install python3-smbus
$ sudo apt-get install python3-pip
$ sudo apt-get install python3-dev
```

Python コマンドで Python3 を起動できるようにしておきましょう。alias を使って、python コマンドを再定義します。バージョンを表示してみると 2.7 から 3.7 に変わっています。

```
$ python -V
Python 2.7.16
$ alias python='/usr/bin/python3'
$ python -V
Python 3.7.3
```

ログインしたときに alias が有効になるよう、/home/chuji/.bashrc の末尾に、次の一行を追加するのが一般的です。

```
export alias python='/usr/bin/python3'
```

ただし、この方法ではユーザ名 chuji でログインしたときしか有効になりません。システム立ち上げ時に自動的に実行するときは、別の手当てが必要です。そこで/usr/bin/python のシンボリックリンクを変更して、python3 を実行するようにします。リンク作成コマンド ln のオプション-f は、既に存在しているリンクを強制的に書き換える指示です。ln コマンドの詳細な説明は省略します。

```
$ python -V
Python 2.7.16

$ ls -l /usr/bin/python*
lrwxrwxrwx 1 root /usr/bin/python -> python2
lrwxrwxrwx 1 root /usr/bin/python2 -> python2.7
-rwxr-xr-x 1 root /usr/bin/python2.7
lrwxrwxrwx 1 root /usr/bin/python3 -> python3.7
-rwxr-xr-x 2 root /usr/bin/python3.7

$ sudo ln -f /usr/bin/python3 /usr/bin/python
[sudo] chuji のパスワード:

$ ls -l /usr/bin/python*
lrwxrwxrwx 2 root /usr/bin/python -> python3.7
lrwxrwxrwx 1 root /usr/bin/python2 -> python2.7
-rwxr-xr-x 1 root /usr/bin/python2.7
lrwxrwxrwx 2 root /usr/bin/python3 -> python3.7
-rwxr-xr-x 2 root /usr/bin/python3.7

$ python -V
Python 3.7.3
```

上の画面を見やすくするように、ls -l の出力表示の一部を削除しています。

NODE.JS

Node.js は、Web サーバーを JavaScript で実現するためのパッケージです。残念ながら Raspberry Pi 用のパッケージサイトで提供されているのは Raspberry Pi 2 以降で使われている CPU 向けなので、Raspberry Pi ZERO や初期の Raspberry Pi では、apt-get による標準的なインストール方法が使えません。直接 Node.js Foundation から最新版をダウンロードしました。他の CPU についても以下の armv6l を armv7l とか arm64 に置き換えれば、同じようにインストールできます。wget や tar の詳細な説明は省略しています。

なお、この方法はグローバルインストールといって、すべてのディレクトリから Node.js を使えるようにしています。この外にローカルインストールといって、自分のホームディレクトリ下にインストー

ルして、同じコンピュータ上で異なるバージョンの Node.js を使うようにもできます。

まず https://nodejs.org で Node.js の安定バージョン（LTS: long term support）を調べます。この時点では、armv6l 用のパッケージで LTS ステータスにあったのは 10.19 でした。PC のブラウザで、https://nodejs.org/dist/にアクセスします。ディレクトリ一覧が表示されるはずです。ここで latest-v10.x/のディレクトリ下にあるファイルのうち、名前の最後の方が linux-armv6l.tar.gz となっているファイルをダウンロードします。なお、armv6 の次の文字は数字の 1 ではなく、アルファベットの l（エル）なので間違えないようにしてください。この本の執筆時点では node-v10.19.0-linux-armv6l.tar.gz が最新でした。ダウンロードしたものを、smb 経由で Raspberry Pi（私の場合はホームディレクトリ）に転送します。または、Raspberry Pi に SSH 接続して以下のコマンドを実行しても同じことです。

```
$ wget https://nodejs.org/dist/latest-v10.x/node-v10.19.0-linux-armv6l.tar.gz
```

このファイルを tar コマンドで解凍し、/usr/local/bin に実行ファイルをコピーします。cp コマンドの-R オプションは、その下のディレクトリを含めてコピーするという意味です。

```
$ tar -zxvf node-v10.19.0-linux-armv6l.tar.gz
$ cd node-v10.19.0-linux-armv6l
$ sudo cp -R * /usr/local
```

まだ/usr/local/bin のファイルが実行できない（パスが通っていない）と思うので、以下のコマンドを実行しておきます。~/.bashrc の末尾に同じ内容を追加しておけば、ログインしたときから node などが使えるようになります。

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

念のため、バージョンを確認しておきましょう。

```
$ node --version
V10.19.0
$ npm -version
6.13.4
```

Node.js のバージョンがあまり新しいと、次の Socket.IO のインストールができないときがあります。その時は、上の手順で Node.js のバージョンを古いものと交換してみてください。LTS なら大丈夫だとは思いますが。

Socket.IO

Socket.IO パッケージは、ブラウザと Web サーバーとの間で通信をするためのパッケージです。通信に使えるプロトコル（通信規約）は何種類もあるのですが、Socket.IO は最適のプロトコルを選んでくれ、統一的なインターフェースから使うことができます。前のプロジェクトでも採用しました。

Socket.IO は npm（Node.js 用のパッケージ管理ソフトウエア。Node.js と同時にインストールされている）を使ってインストールします。-g オプションは、グローバルインストールを指定しています。

```
$ sudo npm install -g socket.io
+ socket.io@2.3.0
added 41 packages from 33 contributers in 31.914s
```

Redis

Redis（REmote DIrectory Server）はメモリ上にデータベースを作るためのパッケージです。ここでは 、そのごく一部である FIFO（First-In-First-Out）機能を使って、プロセス間で通信するために使います。同じコンピュータ上のプロセス同士で通信をする方法は、これ以外にもありますが、この方法が早くて便利なので、前のプロジェクトから採用しました。

まず、Redis サーバーをインストールします。この段階でサーバーは自動的に起動されるようになります（いまの Raspbian では自動的にインストールされるので、実際には確認するだけです）。

```
$ sudo apt-get install redis-server
```

次に、Python と JavaScript から Redis を使うためのパッケージをそれぞれインストールします（ここでも Python3 用の Redis クライアントは既にインストールされていました）。

```
$ sudo pip3 install redis
$ sudo npm install -g redis
```

今回は C 言語から Redis を使うためのクライアントパッケージ hiredis もインストールします。まずリポジトリからコピーを得る git のインストールから始めます。次に make を使ってコンパイルし、make install で必要なディレクトリにコピーします。make は多くのファイルをコンパイルするための道具で、処理するファイル名や具体的な処理内容は同じディレクトリにある Makefile というファイルに記述されています。

hiredis はインストールが間違っていても、自動的にアンインストールすることができません（Makefile に必要な情報が書かれていればできるのですが）。そのため、make -n install（実際にインストールは行わず、どこに入れるかという情報だけを出力する）で、インストール情報をテキストファイル hiredis.log に保存しておきます。最後に make install でインストールを実行します。アンインストールする必要が出てきたら、hiredis.log を読みながらファイルやディレクトリを削除することができます。

```
$ sudo apt-get install git
:
$ git clone https:/github.com/redis/hiredis.git
$ cd hiredis
$ make
$ make -n install >hiredis.log
$ sudo make install
```

インストールは済みましたが、Linux のリンカー（コンパイルしたプログラムのオブジェクトコードをリンクして、実行可能にする）ld は、まだ新しいライブラリがインストールされたことを知りません。コマンド ldconfig で必要なリンクを自動的に作らせます。

```
$ sudo ldconfig
```

あとは、コンパイル時にインクルードファイルを探すディレクトリ（-I オプション）と、hiredis のライブラリをリンクする（-l オプション）を指定してやります。

```
$ cc  -o  <実行可能ファイル名>  <ソースファイル名>
-lhiredis  -I/usr/local/include/hiredis
```

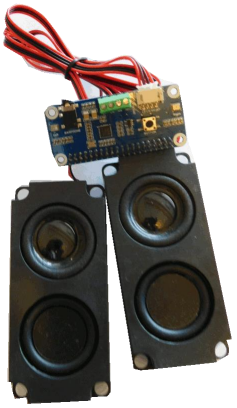
3.3　オーディオユニット

小型のオーディオユニットとして、Waveshare 社製の WM8960 Audio HAT を使うことにしました。ステレオの入出力があり、主な仕様は下表のとおりです。GPIO40 ピンコネクタで Raspberry Pi ZERO の上に重ねることができます。手に入らなければ、類似の仕様のもので置き換えてください。

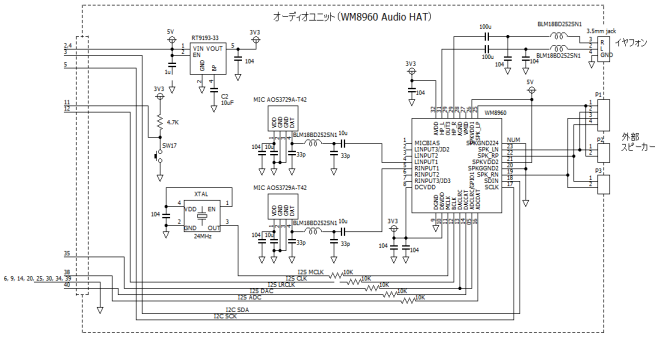
項目	仕様
供給電源	DC 5V
CODEC チップ	WM8960
制御インターフェース	I2C バス
オーディオインターフェース	I2S バス
オーディオスピーカー出力	1W/8Ω×2
オーディオマイク	AOS3729A-T42×2

WM8960 Audio HAT の仕様

ハードウエアとしては、Cirrus Logic 社の CODEC チップ、WM8960 を採用しています。アナログ入力をデジタル変換して取り込み、再生デジタルデータをアナログ信号にして D 級アンプで増幅・出力しています。この HAT には、プラスチックモールドされたスピーカーが一組付属しているので、お買い得だと思います。



マイクは Sanico Electronics 社製の MEMS（微細機械式）マイク AOS3729A-T42 を 2 個搭載しています。プリント板上で離してあるので、ステレオ受信すれば音源の方向が分かるのかもしれません。しかしロボット本体の横幅が狭いので、上下方向に配置したうえで一方のみを使うことにしました。回路図を次に示します。



オーディオユニットの回路図

オーディオ出力には、ステレオスピーカーに接続する端子板、コネクタ、イヤフォンジャックがあります。ここでは、スピーカーに付属しているコネクタケーブルを使いました。

WM8960 のドライバーを Waveshare 社のサイトからインストールします。先にインストールした git を使ってソースのコピーを取得し、インストール用のシェルスクリプト install.sh を起動します。

```
$ sudo apt-get install git
$ git clone https://github.com/waveshare/WM8960-
Audio-HAT
$ cd WM8960-Audio-HAT
$ sudo ./install.sh
```

インストールにはかなり時間がかかります。じっと待って、終了したらリブートします。Seeed 社のユニットを使う場合は、上の URL を https://github.com/respeaker/seeed-voicecard.git

に、作業ディレクトリを seeed-voicecard に読み替えてください。

立ち上がったら、まず HAT が認識されているか調べます。再生ソフト aplay -l で使用可能なオーディオデバイスのリストを取ってみます。ちなみに、ハードウエアを操作するので、コマンドを sudo で実行してください。

```
$ sudo aplay -l
**** ハードウェアデバイス PLAYBACK のリスト ****
カード 0: wm8960soundcard [wm8960-soundcard], デバイス 0: bcm2835-i2s-wm8960-hifi wm8960-hifi-0
[bcm2835-i2s-wm8960-hifi wm8960-hifi-0]
  サブデバイス: 1/1
  サブデバイス #0: subdevice #0
カード 1: ALSA [bcm2835 ALSA], デバイス 0: bcm2835 ALSA [bcm2835 ALSA]
  サブデバイス: 7/7
  サブデバイス #0: subdevice #0
  サブデバイス #1: subdevice #1
  サブデバイス #2: subdevice #2
  サブデバイス #3: subdevice #3
  サブデバイス #4: subdevice #4
  サブデバイス #5: subdevice #5
  サブデバイス #6: subdevice #6
カード 1: ALSA [bcm2835 ALSA], デバイス 1: bcm2835 IEC958/HDMI [bcm2835 IEC958/HDMI]
  サブデバイス: 1/1
  サブデバイス #0: subdevice #0
カード 1: ALSA [bcm2835 ALSA], デバイス 2: bcm2835 IEC958/HDMI1 [bcm2835 IEC958/HDMI1]
  サブデバイス: 1/1
  サブデバイス #0: subdevice #0
```

WM8960 がカード 0、デバイス 0 として認識されていることが分かります。これ以後は hw:0,0 または plughw:0,0 として指定できます。

以下の試験では使いませんが、他のソフトウエアでは ALSA (Advanced Linux Sound Architecture)という Linux 標準のドライバーを使います。以下のコマンドで ALSA に登録しておきます。コマンド最後の store は、設定結果をファイルに保存して以後も使えるようにするための指示です。

```
$ sudo alsactl --file=/etc/wm8960-soundcard/wm8960_asound.state store
```

ALSA 用のツール類もインストールしておきます。

```
$ sudo apt-get install alsa-base alsa-tools
```

いったんリブートしてから、ハードウエアの検証に進みます。

3.3.1音声再生

まず音声を再生してみます。PC で適当な wav 音声ファイルを探し、Raspberry Pi に転送しておきましょう。オプション-D はデバイスの選択に使います。

```
$ sudo aplay -D hw:0,0 test.wav
```

test.wav ファイルの音声が再生されたら OK です。音声ファイルがモノラルの場合は、デバイスの選択を plughw:0,0 とします。

```
$ sudo aplay -D plughw:0,0 test.wav
```

3.3.2音声録音

次にマイクからの音声をファイル test1.wav に録音してみます。コマンドは arecord です。オプションの-f はサンプルフォーマットで、符号付 32 ビット、リトルエンディアン（下位バイトが先に来る）にしました。 -r はサンプリングレート（sps）で通常は 16ksps、-c は入力チャンネル（マイク）を指定します。-d は録音時間（秒）を指定します。指定しない場合は、ずっと録音が続いているので、ctrl-C キーから割り込みをかけて止めます。

```
$ sudo arecord -D hw:0,0 -f S32_LE -r 16000 -c 1 -d 3 test1.wav
```

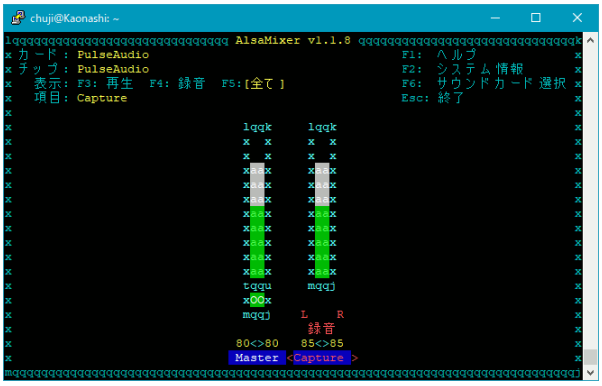
録音した音声を再生してみましょう。

```
$ sudo aplay -D plughw:0,0 test1.wav
```

録音と再生が確認できたら、とりあえずハードウエアの検証を終えます。

3.3.3音量の調整

オーディオユニットの録音・再生音量調整には、alxamixer というコマンドを使い、バーグラフを表示させながら調整するのが一般的です。



しかし、システム起動時にロボットを自動的に立ち上げるには、コマンド型の処理が必要です。それには amixer というコマンドを使います。例えば、録音音量を 80%、再生音量を 100%にするには、以下

のような二つのコマンドを実行します。システム起動時に実行される rc.local に記述しておけば便利です。

```
$ amixer sset Capture 80%
$ amixer sset Master 100%
```

後で出てきますが、Raspberry Pi ZERO に搭載されているオンボード・オーディオは使わない設定にしているので、上でもカードを指定していません。カードを指定するなど、amixer の他の機能については、別の資料を参照してください。

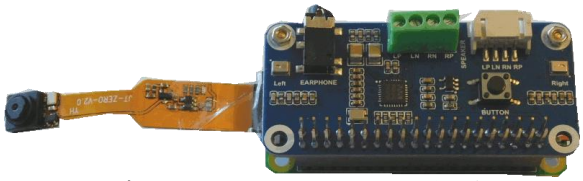
3.4　カメラユニット

カメラとしては、OminiVision 社製 1/4 インチ・5M ピクセルのカラーCMOS 撮像素子 OV5647 を使ったものを手に入れました。この外にもソニー製の 8M ピクセル撮像素子 IMX219PQ を採用したユニットもあります。大事なことは、Raspberry Pi ZERO 用のケーブル（付属または別売）を手に入れることです。カメラの仕様を以下に示します。

項目	仕様
撮像素子	OV5647
撮像素子サイズ（光学）	1/4 インチ
解像度（素子数）	QSXGA（2592×1944）
最大撮影レート	15～120fps（座像サイズによる）
視野角	72.4 度

半導体撮像素子には CCD と CMOS の二種類があります。CCD は感度が高く高解像度で鮮明な画像が得られ、デジタルカメラに多く採用されています。CMOS は過去にはノイズが多いなどの問題があったものの、いまは改善が進み、小型低価格のためスマートフォンなどで多く採用されています。

フレキシブルケーブルを Raspberry Pi ZERO のカメラコネクタに接続します。このときケーブル抑えに無理に力を加えるとプラスチック部品が折れてしまうので、気を付けてください。



オーディオユニットとカメラユニットを組み付ける

ドライバーソフトは標準装備なので、raspi-config でカメラを使えるようにするだけで、すぐテストできます。おっと、その前に、カメラレンズに貼ってある保護フィルムを外してくださいね。右図に示したレンズの周りの凸凹を回して、ピントを調整できます。

3.4.1静止画撮影

静止画撮影コマンド raspistill を実行します。オプション-t は撮影までの待ち時間（ミリ秒）、 -o は撮影画像を保存するファイル名（jpg 形式）、-w と-h は画像の横幅と高さ（単位は画素数）です。-w と-h を省略すれば、最大画素数で保存されます。

```
$ sudo raspistill -t 10 -o test.jpg -w 800 -h 600
```

Raspberry Pi にはディスプレーをつけていないので、test.jpg を直接見ることができません。PC に転送してから確認してください。

出荷時にカメラのピントは無限遠にあっているはずです。あまり近くを撮影するとピンぼけになってしまいます。レンズ周辺に凸凹になった部分があり、ここをピンセットなどで回転させれば、ピントの調節ができます。いちいち PC へ転送してから見るのは面倒ですね。秘書ロボットのカオナシ Web サーバーを動作させれば、画像を見ながら調整できます。

3.4.2動画撮影

動画撮影コマンド raspivid でビデオファイルを作ります。オプション-t は撮影時間（ミリ秒）、-o は出力ファイル名です。ここの拡張子は h264 です。手ブレしやすいので、注意してください。

```
$ sudo raspivid -o test.h264 -t 3000
```

h264 は、ちょっと特殊な形式なので、そのままでは再生できないことが多いようです。私は Windows PC に AnyVideoConerter というソフトを入れてあり、これで mp4 ファイルに変換してから見ていました。

Raspberry Pi 上で変換できるようなソフトをインストールしたほうが便利です。

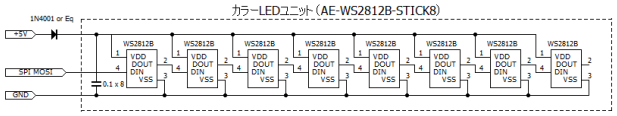
```
$ sudo apt-get install gpac
:
$ MP4Box -fps 30 -add test.h264 test.mp4
```

MP4Box のオプション-fps は動画のフレームレート（一秒間のコマ数）、-add は test.h264 を入力ファイルにすることを指定しています。変換結果は test.mp4 に保存されます。

できあがったファイルは、PC 上で再生することができます。ここまで進めば、ハードウエアの検証は終わりです。

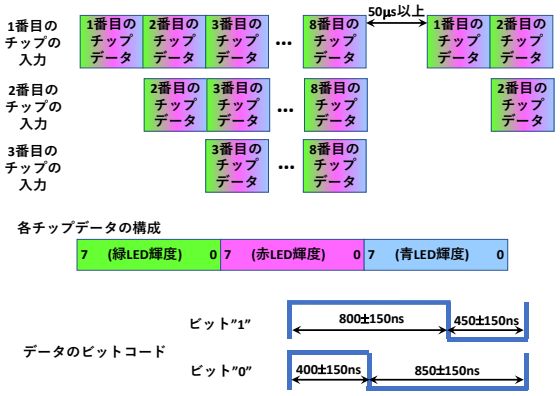
3.5　LED ユニット

ロボットの状態を 3 色カラーLED で表現します。先に説明した 8 チップ搭載のモジュールを使用します。



LED ユニット回路図

チップは直列に接続し、8 ビット×3 色×8 チップのデータをシリアル変換して送ってやると、最初のチップが自分用のデータ（24 ビット）を取り出して、残りを次のチップに送ってくれます。



カラーLED WS2812B データ送信タイミング

各色の輝度はおのおの 8 ビットのデータで表され、各ビットは非対称な波形でエンコードされています。1 ビットの転送にかかる時間は 1.25μs で、800kbps のシリアル通信になります。

チップのピン配置と電気的仕様は次の表のとおりです。LED 駆動電流など、一部の電気的仕様は、あまり明確になっていません。

ピン番号	信号名	説明と仕様
1	V_{DD}	電源（3.5～5.3V）
2	D_{OUT}	次段へのデータ出力
3	V_{SS}	グランド
4	D_{IN}	データ入力（$0.7V_{DD}$ /$0.3V_{DD}$）

LED チップ（WS2812B）のピン配置と電気的仕様

ここで問題になるのは、電源電圧と信号レベルです。Raspberry Pi の 3.3V 電源電圧は仕様外です。一方 5V で駆動すると、Raspberry Pi のハイレベル出力は $0.7V_{DD}$ に達しません。可能な選択肢は以下のとおりです。

1. 3.3V 電源で動かす（不足だが、大体は動く）
2. 5V 電源で動かし、GPIO で直接駆動する（ハイレベル電圧が不足するが、大体は動く）
3. 5V 電源で動かし、GPIO 信号レベルを他の素子（74HCT125 など）で 5V に変換する
4. 5V 電源を 4.7V 以下に降圧して使う

選択肢 1 が一番簡単だし、次は 2 でしょう。ふつう、仕様には余裕（マージン）があるので、仕様外でも動くことが多いものです。しかし、マージンを食いつぶすような「仕様外」の使いかたは感心しません。そこで、選択肢 4 をとることにします。電源電圧範囲はかなり広いので、安定化電源を作るまでもありません。整流用ダイオードの順方向電圧降下（0～1A 流したとき 0.6～1.1V 程度）で充分です。左の回路図を見てください。

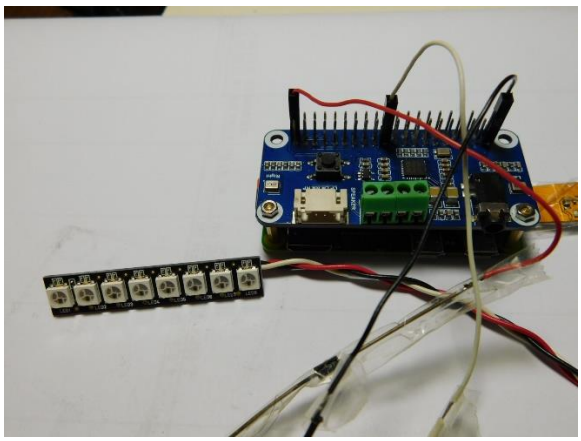
通信速度は 800kbps なので、ハードウエアの助けが必要です。特殊な信号なので、他の用途とは共用できません。Raspberry Pi には、1 ビットシリアル通信に使えるハードウエアが 3 種類あります。

a. パルス幅変調（PWM）が 2 回路
b. シリアル・ペリフェラル・インターフェース（SPI）通信が 1 回路
c. パルスコード変調（PCM）が 1 回路

システム LSI に搭載されている回路数はもっとあるのですが、GPIO コネクタに接続されている数を示しました。

インターネット上では、PWM を使った例が多く説明されています。しかし、秘書ロボットでは a と c はオーディオユニットとのインターフェース（I2S）に専有されてしまっているので、選択肢 b の SPI 通信を採用するしかありません。Raspberry Pi から一方的に送信するので、SPI MOSI（master-out, slave-in）ピンだけを使います。1 ビットの伝送時間（1.25μs）を三分割して、417ns ごとに 0/1 のパターンをシリアル伝送（3×800kbps = 2.4Mbps）すれば、良いことが分かります。

本体に組み込む前に、次ページの写真のようにジャンパーピンなどで仮配線した状態で試験します。短絡に注意してください。



仮配線で動作を確認する

次に、Python 用のドライバーソフトを組み込みます。

```
$ sudo pip3 install rpi_ws281x
:
Successfully installed rpi-ws281x-4.2.3
```

このソフトウエアの動作を確認するためのテストプログラム（https://github.com/rpi-ws281x/rpi-ws281x-python/blob/master/examples/strandtest.py）をダウンロードし、以下の修正（赤字）を行います。LED の数は 8 個、SPI MOSI（GPIO10）を制御に使います。LED_BRIGHITNESS は LED を明るくしすぎないための係数で、LED の明るさ設定値（0～255）に掛けて、実際に設定する値を小さくします。255 ではあまりに明るすぎるための処置ですが、実際のロボットでは使いません（255 に設定します）。

strandtest.py（一部のみ掲載。赤字部を修正する）

```
# LED strip configuration:
LED_COUNT = 8          # Number of LED pixels.
LED_PIN = 10           # GPIO pin connected ...
# LED_PIN = 10         # GPIO pin connected to...
LED_FREQ_HZ = 800000   # LED signal frequency ...
LED_DMA = 10           # DMA channel to use for ...
LED_BRIGHTNESS = 12    # Set to 0 for darkest ...
LED_INVERT = False     # True to invert ...
LED_CHANNEL = 0        # set to '1' for GPIOs ...
```

テストプログラムを実行する前に、Raspberry Pi ZERO の設定をしておきます。まず、転送用データのサイズを大きく取っておきます。以下の二つのファイルを編集するときには sudo が必要です。

/boot/cmdline.txt（赤字部を追加する）

```
console=serial0,115200 console=tty1
root=PARTUUID=56cd6262-02 rootfstype=ext4
elevator=deadline fsck.repair=yes rootwait
spidev.bufsiz=32768
```

次に GPU のクロックを 250MHz に下げておきます（デフォルトは 400MHz）。こうしておかないと、シリアル通信の転送速度がだんだんズレていってしまうからです（インターネット上の説明は見つけにくいし、不明瞭なのが困ったことです）。このクロックはメモリアクセスにも使うので、CPU のパフォーマンスが落ちてしまうのですが、やむをえません。CPU ボード上のオーディオと干渉するという情報があったので、念のため snd_bcm2835 をオンにする記述をコメントアウトしました。これでオーディオ用ハードウエアは WM8960 だけが見えるようになり、カードを指定しなくてもよくなります。

/boot/config.txt（一部のみ掲載。赤字部を追加する）

```
（略）
# Enable audio (loads snd_bcm2835)
# dtparam=audio=on
（略）
[all]
#dtoverlay=vc4-fkms-v3d
start_x=1
gpu_mem=128
dtoverlay=i2s-mmap
dtoverlay=wm8960-soundcard

core_freq=250
```

ファイルの修正が終わったら、リブートして設定を有効にします。それからテストプログラムを実行すれば、さまざまなパターンで発光することが確認できます。

```
$ sudo python strandtest.py
Press Ctrl-C to quit.
Use "-c" argument to clear LEDs on exit
Color wipe animations.
Theater chase animations.
Rainbow animations.
:
```

テストプログラムを読んでみると、5 つの操作や関数を使っていることが分かります。

strandtest.py（一部のみ掲載）

```
（略）
# Define functions which animate LEDs in various
ways.
def colorWipe(strip, color, wait_ms=50):
    """Wipe color across display a pixel at a
time."""
    for i in range(strip.numPixels()):
        strip.setPixelColor(i, color)
        strip.show()
        time.sleep(wait_ms / 1000.0)
（中略）
# Main program logic follows:
（中略）
# Create NeoPixel object with appropriate
configuration.
    strip = PixelStrip(LED_COUNT, LED_PIN,
LED_FREQ_HZ, LED_DMA, LED_INVERT, LED_BRIGHTNESS,
LED_CHANNEL)

# Intialize the library (must be called once before
other functions).
    strip.begin()
（中略）

            colorWipe(strip, Color(255, 0, 0))
（後略）
```

それぞれの機能を次の表にまとめておきます。ロボットの感情表現にはこれだけで充分です。

操作	機能（処理内容）
PixelStrip(……)	オブジェクトを生成する （ハードウエアが初期化される）
.begin()	内部データを初期化する
Color(r, g, b)	三色データを内部形式に変換する
.setPixelColor(i, c)	i 番目の LED に色 c を設定する
.show()	色データを LED に転送する

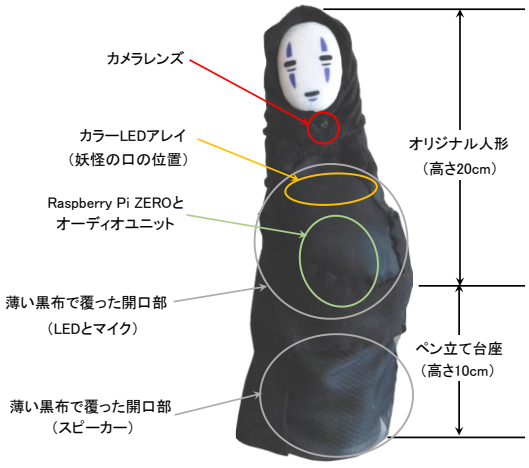
LED の制御に使うオブジェクト

3.6　電源ユニット

ロボット全体が DC5V で動作するので、5V の AC アダプターを用意します。消費電流は 1A にも満たないので、携帯電話用の小型（USB インターフェース）で良いと思います。ただし、ロボット側のコネクタをマイクロ USB にすると、ケーブルを足に引っ掛けたりしたとき壊れやすい。そこで、他の機器で使われている外径 2.1mm の DC プラグに置き換えました。

基台にするペン立ての後ろ側に、DC 入力コネクタと電源スイッチを取り付けました。そう言えば、C3-PO には、首の後ろにスイッチがありましたね。

DC 入力コネクタと電源スイッチ

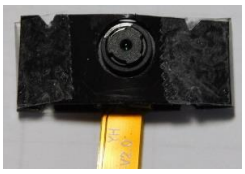


ロボットの最終外形

3.7　ロボット本体の組み立て

ロボット本体として「カオナシ」の人形を使います。この人形は、柔らかい胴体に、衣のような布をかぶせてあります。

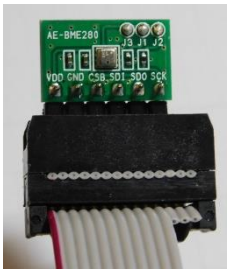
もう使っていない黒いカードに穴を開け（下左）て、切り出した部品でカメラを挟むように加工しました（下右）。凹部を使って、カオナシの顔のすぐ下（の衣）に糸で固定します。完全な固定ではなく、カメラの向き（軸線周りの回転と仰角）は調整できるようにしています。



その下の胴体に LED ユニットを固定します。LED の光を直視しないようにするため、拡散材（白いクリアファイルを切り出したもの）で被いました。

胴体の腹にあたる部分には Raspberry Pi ZERO とオーディオユニットを取り付けます。オーディオユニットから飛び出している GPIO ピンは、厚さを制限するため、切ってしまいました。基板の色が目立たないよう、マイクのところに穴を開けた黒いフェルトを被せました。

実証モデルまでの構想には含まれていませんが、フルモデルで採用する環境（温度、湿度、気圧）センサを I2C バスに取り付けておきます。外形の工作が終わってからでは、加工が難しいからです。右上図の基板中央にあるのがセンサです。誤接触を避けるため、絶縁フィルムで被いますが、開口部を作って空気が流通するようにしました。センサ部をペン立てに入れておきます。



人形の衣の前部分を切り開き、LED ユニットとオーディオユニットが露出するようにします。このままでは不細工なので、洋服の裏地に使う黒いメッシュ様の布（手芸店で相談しました）で塞ぐように縫い付けました。腹部はおおむね黒いものの、LED の発光は見えるし、音がこもることもありません。Raspberry Pi ZERO とオーディオユニットの辺りが、ちょっと出っ腹ですが、それもカオナシらしいと思います。最終外形を左に示します。

人形をペン立ての上に固定し、黒いフェルトで衣を下に延長すれば、身長 30cm のカオナシができ上ります。スピーカーを入れたペン立てはそれなりに重いので、安定感が得られました。

3.8　全回路図

ここまでで、全部のユニットができ上がりました。それらの接続を含めた GPIO 周りの全回路図を下にまとめます。なお、カメラユニットは専用コネクタを使って接続しているいるので、回路図には示していません。

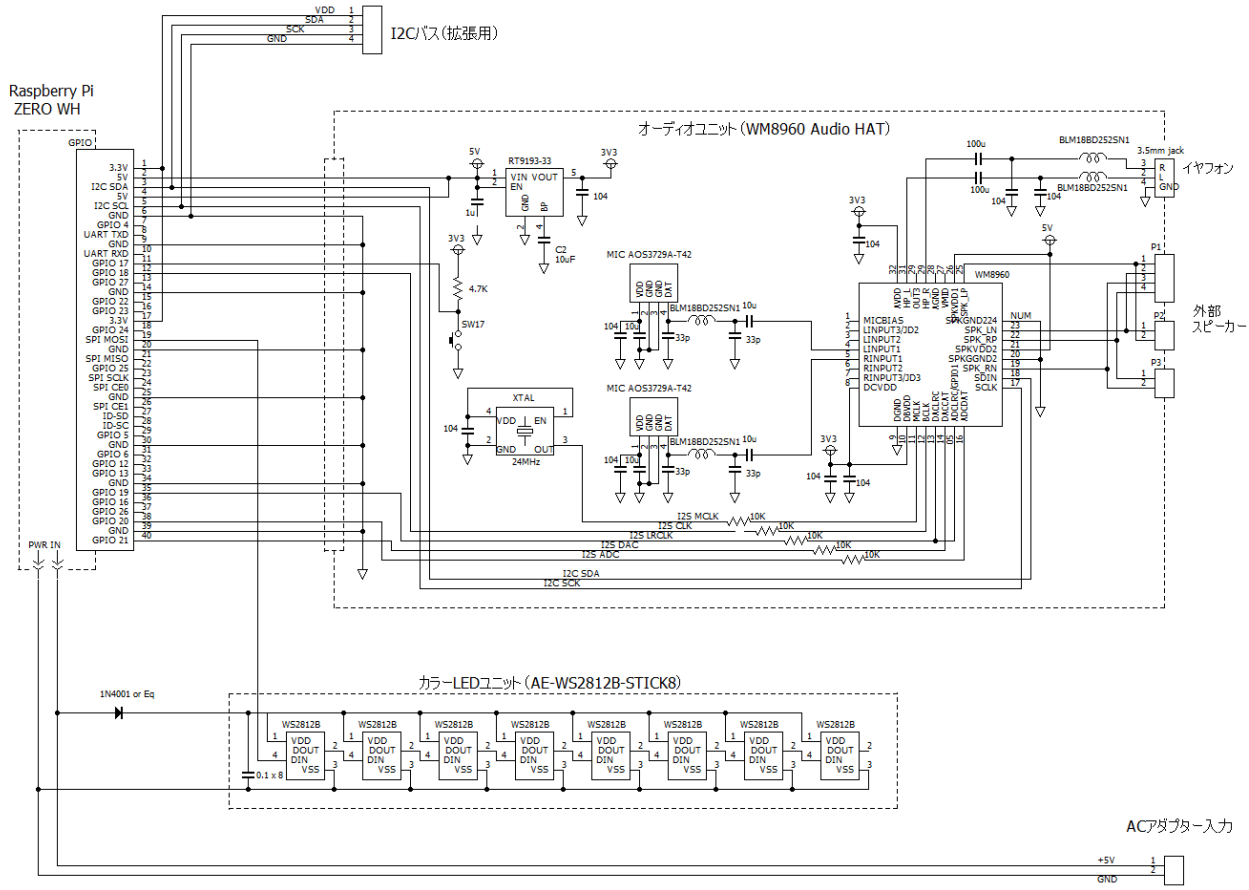
フルモデルまでは使いませんが、I2C バスの拡張部には環境センサが接続されています。

コラム　その後の三原則

AI やロボット技術が発展するにつれ、「開発者の倫理」としての原則も重要になってきました。2016 年、マイクロソフトのサティア・ナデラ CEO は、雑誌のインタビューで「AI 開発の原則」について、こう語っています。

1. AI は人間の自律性を尊重し、人類を助ける目的で開発すべきである
2. AI の作動原理は人に理解できるようにし、人類はそれを理解すべきである
3. AI は人間の尊厳を冒さない範囲で、最大効率化されるべきである
4. AI は人間のプライバシーを尊重し、情報を守ることで人類の信頼を得るようにすべきである
5. AI があやまって危険なことをしようとしたら、人間が止められるようにすべきである
6. AI は偏見を持たず、人間を差別しないようにすべきである

さて、これからどうなるのでしょうか？



秘書ロボット「カオナシ」全回路図（カメラを除く）

4　既存ソフトウエアの組み込み

ロボットの機能のすべてを一から自作しようとすると、開発するソフトウエアの分量は膨大なものになってしまいます。それだけバグも多くなり、望みどおりに動かすのは困難になるでしょう。

そこで、インターネット上で調達できる既存ソフトウエアを、できるだけ使うようにしました。Raspberry Pi ZERO という、あまり処理能力が高くないプロセッサを使っているので、クラウド型サービスも選択肢に含めます。

この章では、フェーズ 2 開発として、既存ソフトウエアの組み込みと、パフォーマンスの評価を行います。Ubuntu を搭載した PC、あるいは Raspberry Pi B3/4 をお持ちの方は、そちらでも試してください。CPU パワーの差が体感できます。

4.1　音声合成

任意の（日本語の）文章を、ロボットに読み上げさせる方法を考えます。それには

1. 音声合成チップを組み込む
2. 音声合成ソフトを組み込む
3. クラウドサービスを使う

の三とおりの実現方法があります。いま使える音声合成チップは、アクエスト社の AquesTalk pico くらいです。ソフトウエア合成の音質が良くなってきたので、チップ化では追いつけないようです。

組込める日本語音声合成ソフトウエアとしては、

A. Open JTalk（名古屋工業大学）
B. VoiceText（HOYA）
C. microAITalk（エーアイ社）
D. FineSpeech（アニモ社）

などがあります。その他にも、英語などの合成なら eSpeak など、軽快に使えるものがあります。英国エジンバラ大学と米国カーネギーメロン大学が開発した Festival も「日本語の合成に使える」と喧伝していますが、実用例は見つかりませんでした。というわけで、無償で使えて実用に耐えるのは Open JTalk だけです。これを組み込むことにします。

クラウドサービスとしては、上のソフトウエアをクラウド化したものの 他に、以下のサービスが代表的です。

a. Amazon Polly
b. Amazon Alexa Voice Service
c. Google Cloud Text-to-Speech
d. Microsoft Azure Cognitive Service: Speech API (Text-to-Speech)
e. 東芝 Coestation

ほとんどが有料なので、音声認識や検索と合わせて、あとで検討することにします。

4.1.1Open JTalk

Open JTalk を組み込んで、音質と合成パフォーマンスを調べることにします。合成するテキストの長さは最大 1024 バイト（UTF-8 日本語で 341 文字）ということですが、このロボットで問題になることはありません。

（この段落と下の操作画面は参考情報です）Open JTalk は apt-get でインストールすることができます。PC 上で Open JTalk を「使ってみる」だけなら、本体と辞書、それに波形生成システムを以下の要領でインストールできます。しかし、後で改造することを念頭に、ソースコードからのインストールを選ぶことにしました。

```
$ sudo apt-get install open-jtalk
$ sudo apt-get install open-jtalk-mecab-naist-jdic
$ sudo apt-get install hts-voice-nitech-jp-atr503-m001
```

各々の要素のインストールは、パッケージを PC でダウンロードし、Raspberry Pi ZERO のホームディレクトリにコピーするところから始めます。それぞれを解凍したら、解凍したディレクトリに移ってコンパイルとインストールを行います。

音声合成エンジンのインストール

音声合成エンジンは、PC を使って次の URL（https://sourceforge.net/projects/hts-engine/files/hts_engine%20API/）から open_jtalk-1.11 .tar.gz をダウンロードし、Raspberry Pi ZERO のホームディレクトリにコピーするところから始めます。

```
$ tar zxvf hts_engine_API-1.10.tar.gz
$ cd hts_engine_API-1.10
$ ./configure
$ make
$ make -n install >install.log
$ sudo make install
```

tar で圧縮ファイルを解凍し、そこに作られたディレクトリの中でコンパイルを行います。

./configure は、比較的規模の大きいソフトウエアのコンパイル・リンクを指示する Makefile ファイルを、環境に合わせて自動生成するツールです。Autotools というパッケージの一部で、Makefile.am ―(automake)→Makefile.in―(configure)→Makefile という順番で Makefile を生成することができます。make は Makefile の指定に従って、コンパイル・リンクを実行します。

次に出来上がったファイルを make install でインストールします。

Open Jtalk 本体

形態素解析エンジンを含む Open JTalk 本体は、https://ja.osdn.net/projects/sfnet_open-jtalk/releases/ から hts_engine_API-1.10.tar.gz をダウンロードします。ホームディレクトリに戻ってから始めます。./configure のオプションでは、音声合成エンジン用のインクルードファイルとライブラリのあるディレクトリを与え、日本語のコードに UTF-8 を指定します。

```
$ tar zxvf open_jtalk-1.11.tar.gz
$ cd open_jtalk-1.11
$ ./configure --with-hts-engine-header-
path=/usr/local/include --with-hts-engine-library-
path=/usr/local/lib --with-charset=UTF-8
$ make
$ make -n install >install.log
$ sudo make install
```

日本語辞書

次に最新の辞書を http://open-jtalk.sourceforge.net/ から PC を使って入手します。Dictionary と書かれたところから Binary Package(UTF-8)をクリックしてダウンロードしたら、Raspberry Pi に転送しておきます。この本を書いたときはバージョン 1.11 が最新でした。Raspberry Pi ZERO のホームディレクトリで解凍し、辞書ディレクトリにコピーしておきます。

```
$ tar zxvf open_jtalk_dic_utf_8-1.11.tar.gz
$ sudo cp -r open_jtalk_dic_utf_8-1.11
/var/lib/mecab/dic
```

声色指定（音響データ）

合成する音声の声色は、Open JTalk の-m オプションで指定できます。/home/chuji/open_jtalk-1.11/の下に htsvoice というディレクトリを作り、ここに音響データを記録している.htsvoice ファイルを集めました。例えば、以下の大学から公開されています。ほかにも.htsvoice で検索すれば出てくるし、自作することもできます。

- 名古屋工業大学（https://sourceforge.net/projects/mmdagent/files/MMDAgent_Example/）
- 東北大学（https://github.com/icn-lab/htsvoice-tohoku-f01/）

集めた.htsvoice ファイルを-m オプションで指定しながら、声の違いを確認することができます。カオナシには、takumi_sad という、ちょっと暗い声が似合いそうです。

音声合成テスト

これで準備ができました。試しに簡単な文を音声にしてみましょう。まず、message.txt に“こんにちは”というテキストを保存します。これを読み上げさせて、音声ファイル message.wav に保存しましょう。下の例では明るい声色 takumi_happy を使っています。。そのファイルを aplay で再生します。

```
$ echo こんにちは >message.txt
$ open_jtalk -m /home/chuji/open_jtalk-
1.11/htsvoices/takumi_happy.htsvoice -x
/var/lib/mecab/dic/open-jtalk/
open_jtalk_dic_utf_8-1.11 -ow ./message.wav
<message.txt
$ sudo aplay message.wav
```

Open JTalk の主なオプションは、この節の最後にまとめてあります。

スピーカーから「こんにちは」という声が聞こえましたか？　Open JTalk は読み上げファイル名を省略すると、標準入力から読み取ろうとします。Linux の標準出力を次段の標準入力に接続する「パイプ」（"|"）を使うと、上と同じことが一行で実行できます。

```
$ echo こんにちは | open_jtalk -m
/home/chuji/open_jtalk-
1.11/htsvoices/takumi_happy.htsvoice -x
/var/lib/mecab/dic/open-
jtalk/open_jtalk_dic_utf_8-1.11 -ow /dev/stdout |
sudo aplay -D plughw:1,0
```

Raspberry Pi ZERO の場合、上のコマンドで音声が出てくるまでに 2~3 秒かかります。三つのコマンド（echo, open_jtalk, aplay）を立ち上げて初期化するのにかかる時間（オーバーヘッド）を考えても、ちょっと時間を食いすぎる気がします。PC で同じことを実行すると、ほとんど時間を意識しないで済むので。ロボットに実装するときには、このオーバーヘッドを減らすのと、何度も使う言葉は、あらかじめ.wav ファイルに変換しておくといった工夫が要りそうです。

下に Open JTalk の主なオプションの一覧を整理しておきます。

オプション	指定内容
-x	辞書を置いてあるディレクトリ
-m	合成に使う音響データ（.htsvoice）ファイル
-ow	出力（.wav）ファイル名
-r	発音速度（0 以上、デフォルト 1.0）
-g	発話の音量を上げる（0db 以上、デフォルト 0db）

Open JTalk の主なオプション

区切り文字（句読点など）

このままだと棒読みに近いので、息継ぎを表現するための区切り文字を、言葉の間に入れてみます。詳しい説明のある文献が見つからなかったので、実験で試してみました。その結果は、次のとおりです。

- 文中（単語の間）に区切り文字を入れると、約 0.5 秒の静音時間が挿入される
- 同じ場所に区切り文字を二つ以上続けても、静音時間は変わらない（同じ区切り文字でも、異なっていても変わらない）
- 文末に区切り文字をつけても、静音時間は付け加えられない（同じ音声ファイル）
- 改行記号が含まれると、その前までしか合成しない。

半角のコンマとスペース、全角のスペースのほか、句読点やカッコなどは、すべて区切り文字として認識されるようです。

0.5 秒より長い息継ぎは、無音の音声ファイルを用意するか、他の処理による遅れを利用することにします。

4.1.2クラウドサービス

Microsoft Text-to-Speech を使ってみました。Web から試せます。Open JTalk より、日本語のイントネーションが自然な気がします。英語の場合は、本当の人間ではないかと思うほど自然な音声になっています。

4 種類のサンプルの声色はどれも明朗で、聞きやすいと思います。あまり妖怪くさくないのが気になりました。サンプル音声（最短でも 30 分！）から声色を作り出してもくれるのですが、声優ではないので特異な声色を出す自信はありません。

外観など秘書ロボットのイメージを考え直すなら、サービス標準の声色で充分実用的です。それでも、しばらくは妖怪カオナシに固執したいと思います。

4.2　音声認識

ロボットに日本語で話しかけ、Siri や Alexa を念頭に（いかにも）知的な反応を引き出す方法を考えます。一時期 codama という「音声対話開発キット」使えるのではないかと調べたのですが、購入できるボードは録音しかできず、NTT ドコモのクラウド（研究開発用途はサービスを停止した）で処理するものでした。さすがにチップ化は困難のようです。そこで、選択肢としては、

1. 音声認識ソフトを組み込む
2. クラウドサービスを使う

があります。しかし、無償あるいは安価で組込める日本語音声認識ソフトウエアとしては、

A. Julius（名古屋工業大学）

が現在では唯一の選択肢のようです。音声を「音素」に分解してから解析するので、処理の重さが気になります。しかし、ほかに選択肢がないので、これを組み込んで試すことにしました。英語用なら Kaldi や CMUSphinx なども候補になります。

代表的なクラウドとしては、以下のようなサービスがあります。組込み用と異なり、ニューラルネットワークを使う、最新技術（少なくとも提供者はそう言っている）を採用しています。

a. Amazon Transcribe
b. Amazon Alexa Voice Service
c. Google Cloud Speech-to-Text
d. Microsoft Azure Cognitive Service: Speech API (Speech-to-Text)
e. Docomo 音声認識 API（Powered by アドバンスト・メディア）
f. IBM Watson Speech to Text

ほとんどが有料なので、音声合成や検索と合わせて、後で目的に合うものを検討することにします。

4.2.1Julius

Julius のインストールは少々やっかいです。膨大なソースコードの形で公開されているため、自システムでコンパイルする必要があります。ここでは PC を使って入手してみましょう。

2020 年 2 月現在の最新版はバージョン 4.5 でした。その説明ページ（https://github.com/julius-speech/julius/releases/tag/v4.5）の一番下にあるリンクから、julius-4.5.tar.gz をダウンロードします。

次に、音声を認識するディクテーション（口述筆記）のためのキットを入手します。このページ（https://julius.osdn.jp/index.php?q=dictation-kit.html）から dictation-kit-4.5.zip をダウンロードします。Julius とダウンロードキットのバージョン番号が一致しないと、うまく動かないので注意してください。両方のファイルを Raspberry Pi のホームディレクトリに転送しておきます。

Raspberry Pi に SSH 経由でログインします。最初に、音響ドライバーALSA の開発キットをインストールしておきます。

```
$ sudo apt-get install libasound2-dev
```

次は、解凍して得られた Julius のソースをコンパイルします。手順は Open JTalk の場合と同じです。./configure のオプション--with-mictype=alsa は ALSA デバイスから音声を入力することを指定するためのものです。

```
$ tar zxvf julius-4.5.tar.gz
$ cd julius-4.5
$ ./configure --enable-words-int --with-
mictype=alsa
$ make
$ make -n install >install.log
$ sudo make install
```

ディクテーションキットは解凍しておき、Julius 起動時に置き場所を教えてあげます。

```
$ cd /home/chuji
$ unzip dictation-kit-4.5.zip
```

最後にログイン時に入力に使う ALSA ドライバーを指定できるようにしておきます。ログインディレクトリにある.bashrc を編集し、末尾に

```
Export ALSADEV=plughw:0,0
```

と追加したら、リブートしてください。次からは、このマイクからの音声入力を処理できるようになります。

Raspberry Pi が再起動したら、次のコマンドを試してみます。-C オプションは、ディクテーションキットの設定ファイル（jconf）を二つ指定します。-demo は動作を見るためのオプションです。

```
$ sudo julius -C dictation-kit-4.5/main.jconf -C
dictation-kit-4.5/am-gmm.jconf -demo
 :
 :
<<< please speak >>>
```

<<< please speak >>>と表示されたら、マイクに向かって何か話しかけます。画面上に何回か候補が表示された後、最終的な結果が Sentence1:のあとに表示されます。この段階では、あまり認識精度が良くないかもしれません。とにかく、動作したということを確認すれば大丈夫です。精度を上げる方法は後で考えます。

「カオナシ」、「こんにちは」、「いい天気ですね」、「少し歩きませんか」と話しかけてみました。結果は下のとおりです。聞いたこともないだろう「カオナシ」という言葉は別にして、だいたい間違いなく聞き取れていることが分かります。一回の認識に約 15 秒かかりました。ちょっと長すぎますね。

```
<<< please speak >>>
pass1_best:
sentence1:  。
pass1_best:  今日 は 、
sentence1:  こんにちは 。
pass1_best:  いい 天気 です か
sentence1:  いい 天気 です ね 。
pass1_best:  少し 歩き ませ ん か 。
sentence1:  少し 歩き ませ ん か 。
<<< please speak >>>
```

単語を認識させるより、文にした方が、精度が上がります。前後関係を考慮するためで、これは人間も同じことをしているので、納得がいきます。もともと、講演などの文字起こしを目的としたソフトウエアなので、紋切り型の短い命令は得意ではないのでしょう。話しかけ方に工夫がいりそうです。

ここで使った-demo オプションでは、聞き取った音声を表示していくだけですが、Julius は常駐サーバーとして動作し、結果を TCP/IP 経由で教えてくれるようにもできます。

4.2.2クラウドサービス

前節で Microsoft を試したので、ここでも Microsoft Speech-to-Text を使ってみました。

流石だというのが素直な感想です。短い言葉も、ある程度長い分も、数秒以内に正しく認識できました。文の最初を聞き間違っても、そのあとを聞いてから訂正してくれます。周囲ノイズの影響も受けにくいようです。十分実用的だと感じました。さしあたっては、当初方針どおり Julius を使っていきますが、どこかで乗り換えようと思います。

5　ソフトウエア開発環境

5.1　ソフトウエアの部品化

注：この章は「Raspberry Pi 中級電子工作：温度コントローラ　～設計から製作・検証・応用・保守まで」の説明とほぼ同じ内容です。すでにご存じの方は、次の章に進んでください。

ソフトウエアを作りっぱなしで、使い捨てにするのはたいへんな無駄です。せっかく Raspberry Pi を買い、ソフトウエアを作ったなら、何度でも、また長いこと使えるようにしたいものです。

そのためには、ソフトウエアを把握しやすいくらい小さい単位で作っておき、他のソフトウエアと組み合わせられるようにします。ソフトウエアモジュールは、それぞれ一個のファイルにしておき、使うときに自動で結合させる手法を紹介します。

5.2　Linux ツールの利用

Raspberry Pi で使うオペレーティングシステム（OS あるいは基本ソフトウエアとも呼ばれます）は UNIX を起源とする Linux です。UNIX は、もともとソフトウエア開発をする人たちが便利に使える OS として開発されました。せっかくなので、その機能を開発環境として上手に使いましょう。

5.2.1cpp

cpp（C 言語プリプロセッサー）は Linux の前身である UNIX の C 言語用に開発されたもので、見やすい形のソースコードを、コンパイラが処理しやすい形に変換します。C 言語に特化しているわけではないので、他の言語でも利用できます。その主な機能は、

(1) 他ファイルの取り込み
(2) テキストの置き換え
(3) 条件付き処理
(4) コメントの除去

です。その指示はすべて#で始まる命令（#は必ず行の先頭から始まり、インデントはできません）からなり、その他の部分は指示に従った処理の後、あるいは元のままで出力します。出力先は Linux の標準出力で、ファイルにリダイレクトしたり、パイプ（次節参照）につないで次のプログラムに渡したりできます。今回のプロジェクトでは次のような命令を使います。

#include（他のファイルを取り込む）

#include 命令は、その位置に他のファイルを取り込みます。取り込むファイル名は、命令のすぐ後（同じ行）に指定します。取り込むファイルが、いま開いているファイルと異なるディレクトリにある場合は、それへの相対パスで指定します。

```
#include "include/vision.h"
#include "eye.py"
```

この機能のおかげで、オブジェクト毎にそれぞれのファイルを作り、部品（モジュール）化することができます。

Python を使った経験のある人なら、ここで「ちょっと待って」と言うかもしれません。「それって、import と同じではないか？」というものです。例えば、eye.py のなかに、こんな記述があります。

```
#include "time_keeper.py"
  :
    self.tim = time_keeper(...)
```

これは、import を使って以下のようにも書けるはずです。わざわざ cpp を持ち出す必要はないように思えます。

```
import time_keeper
  :
    self.tim = time_keeper.time_keeper(...)
```

上の考察は正しいのですが、ひとつ重要な制限があります。それは、import されるモジュールが、Python で実行できる形になっている必要があることです。以下に説明するような、#define を使って名前を定義する、#ifdef を使ってモジュールの機能そのものを切り替えるといった強力な操作は、import されるモジュールでは使えなくなってしまいます。上の例では、さまざまな条件下で time_keeper.py を検証する必要があったので、このモジュールでも cpp を使っているのです。

最終的にモジュールの機能仕様が固定でき、検証が済んで、安定したモジュールとして使えるのであれば、import して使うことを考えても良いでしょう。

#define（名前を定義する）

#define 命令は、次に続く名前（マクロ名）をその後ろにある値（置換テキスト）との関連付けを定義します。処理しようとするファイルの中に、その前

に定義された名前が出てくると、定義された値に置き換えます。名前も値もテキストとして記述します。例として次のようなファイル cpp-test.txt を作ってみましょう。以下、この例のようにファイルの内容を示すときは、ファイル名を黄色欄に示すようにします。

cpp-test.txt

```
#define MY_STRING  hello!
#define MY_NUMBER  123

MY_STRING = MY_NUMBER
```

これを cpp に与えると、以下のように置き換えられます。

```
$ cpp cpp-test.txt
hello! = 123
```

名前には英数字とアンダースコアが使えます。プログラムコードの中では、「これはマクロ定義されたものだ」と分かるように、マクロ名をすべて（場合によっては一部のみを）大文字にすることがよく行われます。この本のなかでも、同じ習慣に従っておきます。

こんな定義をする理由として、以下の 4 つがあげられます。

1. 意味不明の数値をなくす
2. 保守性を良くする
3. モジュールの機能を切り替える
4. 関数も定義できる

1. 意味不明の数値をなくす

プログラムの中に数値を埋め込んでいると、それが何を表すのか分かりません。例えば以下のようなコードでは、

```
if mode == 3:
   :
```

この 3 が何かを数えた結果なのか、約束ごととして取り決めたものなのかよく分かりません。そのプログラムを書いた人には分かっているのでしょうが、他の人が読んでも意味が伝わりません。しばらく後になると、プログラムを書いた本人でさえ思い出せなくなります。こういう意味不明の数値を、「なぜか分からない数値だが、それを使うと動作する」という意味で、マジックナンバーと呼んだりします。マジックナンバーをなくすのが、第一の理由です。

2. 保守性を良くする

プログラムは一度書いたら終わりというわけではなく、使い込んでいくうちに手直しが必要になることがあります。この手続きを保守といいます。プログラムに問題が見つかっても、マジックナンバーがあると、どこに手を入れたらいいのか分かりません。後でプログラムを読み直すために、コメントを付けたりするわけですが、デバグにとり紛れたりして付け忘れることが、しばしばあります。

また、同じ意味で使っている数値が複数の個所に現れることもよくあります。特にモジュール間での情報伝達や、通信を行うときには、プログラム内の離れた場所で使われます。その値を変更する必要ができたとき、一方を変更し忘れたために苦労することになります。その点、定義ファイルに#define しておけば、それを取り込むすべてのプログラムが一度に修正できます。

こういった定義を使うプログラムは、一つのファイルとは限りません。そのため、独立した定義ファイルを作り、各プログラムの冒頭で#include するようにします。冒頭に組み込むことから、こういう定義ファイルは「ヘッダーファイル」と呼ばれます。共通して使えるよう、決まったディレクトリに置くことが多く、ファイル名の拡張子を.h とする習慣があります。

ヘッダーファイルのなかに、特定のプログラム言語のコードを入れるのは、望ましくないと言われています。ただ今回は、プログラムの流れをつかむのに関係ないコードは、ヘッダーファイルに含めることがあります。その場合のヘッダーファイルは、他の言語では使えません。それを明示するため、Python コードを含むヘッダーファイルには use-XXX.h という名前を付けています。

2. モジュールの機能を切り替える

プログラムモジュールは、できるだけ汎用に作りたいものです。用途が異なってもソースコードに手をつける必要がなければ、さまざまな用途に使えます。たとえば音声入力サーバーでは、どの音声認識機能を使うか、マクロ定義を使って切り替えたりしました。

また、検証用のコードを埋め込むのにも使えます。

```
#ifdef DEBUG
   print(process)
#else
   pfork(process)
#endif
```

3. 関数を定義する

マクロ名にカッコと仮変数をつけると、関数（マクロ関数）が定義できます。cpp はマクロ関数を定義した式に置き換えてくれます。例えば、以下のようなファイル my_func.py

my_func.py

```
#define MY_SQUARE(x)  x*x
#define IS_LARGER(x, y)  x>y

if IS_LARGER(this, that):
  print (MY_SQUARE(this))
```

を cpp で処理すると、

```
$ cpp my_func.py
if this>that:
   print (this*this)
```

と変換してくれます。これは実際に関数を作っているのではなく、見やすい名前を、実際の数式に置き換えるだけです。数式が複雑になればなるほど、また何回も使えば使うほど、この定義の効果が表れます。この定義を「マクロ定義」、置き換えを「マクロ展開」と呼びます。

#ifdef ～(#else ～) #endif（条件付き処理）

この定義機能を使って、プログラムの一部を選択することができます。#ifdef（if defined）は、それに続くマクロ名が定義されているときだけ次のテキストを残し、定義されていない場合はそっくり削除します。

```
#ifdef OPTION1
  このテキストは、
  OPTION1 が定義されていると残る
  定義されていないと削除される
#else
  このテキストは、
  OPTION1 が定義されていると削除される
  定義されていないと残る
#endif
```

#else 以下はなくても構いませんが、最後は必ず #endif で終わります。定義されていない場合だけ使いたいときには、#ifdef の代わりに#ifndef（if not defined）を使うことができます。また、#ifdef のブロックの中で別の#ifdef を記述することもできます。

ちょっと変わった使いかたですが、一つの定義ファイルを複数のモジュールで#include する場合があります。それぞれのモジュールを評価するときには問題ないのですが、全体をまとめて動かすときには、重複定義を避ける必要があります。そこでインクルードファイルは、以下のような記述をしておきます。

```
#ifndef __THIS_FILE
  定義の記述
  :
#define __THIS_FILE
#endif
```

一回目に取り込んだ時には、定義が読み込まれます。二度目からは__THIS_FILE が定義されているので、空文しか残りません。__THIS_FILE の定義名は、インクルードファイルごとに変えます。先頭にアンダースコアを 2 回続けるのは、そういうマクロ名が他で使われることはないだろうという推測からです。

/ コメント */*

cpp は、"/* "と" */"で囲まれたテキストを、コメントと認識して、すべて削除します。コメントが複数行にわたっていても、また"/* "が行の途中から始まっても構いません。

-D オプション

cpp には、-D というオプションがあります。このオプションを使えば、元のファイルに手を加えることなく、処理の最初に#define することができます。例えば、

```
$ cpp -DRUN_ALL kaonashi.py
```

とすると、kaonashi.py を開く前に、次のような記述があったとみなします。

```
#define RUN_ALL
```

この機能を使って、#ifdef で条件ごとの処理を記述しておけば、cpp 実行時に使うコードを指定してやることができます。モジュールの機能を切り替えるのに、この方法を使えば、ソースコードに手を加える必要がありません。

重複定義を避けるため、-D オプションで定義を変更する可能性のあるマクロ定義は、#ifndef と#endif で囲っておきます。

5.2.2リダイレクトとパイプ

Linux のプログラムで入出力先を指定しないときは、「標準入力」と「標準出力」が使われます。ふつうの状況では標準入力はキーボード、標準出力はコンソールのスクリーンになっています。プログラ

ムを実行するときに、他の入出力先に切り替えることができます。ファイルから入力させるときは、<(ファイル名)とし、出力先をファイルにするには、>(ファイル名)と指定します。

```
$ echo hello!
hello!
$ echo hello! >tmp
$ echo <tmp
hello!
```

二番目のコマンドは echo の出力先を tmp というファイルにする、三番目の命令は入力先を tmp にするという命令です。三番目の命令は cat tmp と同じことです。

あるコマンドの出力を、別のコマンドの入力にする機能を「パイプ」といって、"|" で指定します。以下のような例をよく見かけますね。これは全てのプロセスのうち、tty というテキストを含むものをすべて表示します。

```
$ ps aux | grep tty
```

リダイレクトとパイプは、比較的簡単なコマンド（プログラム）を組み合わせて、複雑な処理を行うのに有効です。

5.2.3クリーンアップ

すこし前に戻りますが、cpp が（標準出力へ）出力するテキストは、あまり読みやすいものではありません。空行が残っているし、処理した内容を#で始まるコメントとして残しています。そのあとをコンパイラなどで処理する分には問題ないし、#で始まる行は python でもコメントとして無視されます。

それでも人が読みやすいようにするため、きれいにするツールを用意しました。これはマクロ展開などが正しく行われているか確認するときなどに役立ちます。次のような python プログラム cleanfile.py を作っておきます。

cleanfile.py

```
# cleanup procedure for cpp output

while True:
  try:
    s = input()
    if s != "":
      if s[0] != "#":
        print (s)
  except EOFError:
    break
```

空文と冒頭が#で始まる行を削除するだけですが、見やすさは格段に良くなります。実際に使うときは、

```
$ cpp program.py | python cleanfile.py > object.py
$ python object.py
```

とするか、あるいは

```
$ cpp program.py | python cleanfile.py | python
```

とすれば、複数のモジュールからなるプログラムを組み合わせて（#include して）実行することができます。

検証途中でキーボードからの入力を必要とするときがあります。このときは実行時のコードをパイプから受け取っていると、標準入力がそちらを向いているので、入力がないというエラーが起きてしまいます。一度実行コードをファイルに入れる、前の方の手順を取ってください。

5.2.4シェルスクリプト

Linux のコマンドひとつひとつは、比較的簡単な機能を実現する場合が多いのですが、コマンドオプションとパイプを使うと、こみいった処理も行えます。ただ、そのたびに長いコマンド列をタイプするのは効率が悪いし、ミスタイプも起こしやすくなります。そこで命令の列をファイルに記述して、このファイルを実行するのがシェルスクリプトです。例えば runpython というファイルに、以下のように記述します。

runpython

```
cpp $1 | python cleanfile.py | python
```

ここで$1 は、一つ目のパラメータを表します。まず、次のコマンドで、このファイルを誰にでも（a）実行できる（+x）ようにします。

```
$ chmod a+x runpython
```

そこで、以下のようなコマンドを与えて、program.py を実行することができます。

```
$ runpython program.py
```

cpp の-D オプションを変えたシェルスクリプトを作っておくと、ロボットの構成を柔軟に変更できます。

もっと複雑なシェルスクリプトを書くこともできますが、それは専門書を参考にしてください。

5.3　モジュール化

各サブシステムを設計するときは、全体の機能を分析して、複数の機能単位に分解していきます。それぞれの機能単位は、全体機能の一部を実現するもので、機能単位間で独立性が高い（機能の重複がなく、その内部だけで機能の説明ができる）ように設計します。必要なら、さらに小さい単位に分解し、全体を部品（機能単位）で組み立てられるようにします。この段階で、各部品の仕様も詳細化していきます。部品はできるだけ汎用的なものにし、他の用途にも利用することを考えます。

こうやって詳細化された部品は、ソフトウエアのモジュールとして開発します。多くのモジュールはPython のオブジェクトとして定義します。オブジェクトには以下の 3 つの要素があります。

要素	説明
クラス	オブジェクトを生成するための定義
属性（プロパティまたはアトリビュート）	オブジェクトの内部変数
操作（メソッドあるいはサービス）	オブジェクトに対して操作をするための関数

オブジェクトモデル

各々のモジュールのオブジェクトとしての定義は、次章以降の各モジュールの最初に示しています。この資料がなくてもプログラムを理解できるよう、同じか、より詳しい内容を各ファイルの先頭にコメントとして記述しています。

属性へのアクセスは、そのオブジェクトに特有の処理が必要なので、必ず操作関数を介して行います。ただ、属性の読み出しだけは、操作を介すると冗長なので、直接参照することもあります。

5.4　プログラムの検証

作成したプログラムには、間違い（バグ）が含まれていることがあるので、実行しながら間違いを正す（デバグする）必要があります。プログラムの開発仕様に従って、それを満たしているかどうかを検証していきます。検証計画と実行は、プログラムの作成と同じくらい時間がかかるものです。この節では、検証のために、あらかじめ考慮しておくことをまとめます。

5.4.1シミュレーション用コード

自分で作ったハードウエアとソフトウエアを組み合わせて使うと、問題が起こったときに、どちらに原因があるか調べるのが難しくなります。そこで、ソフトウエアのできるだけ多くの部分を、PC 上で検証します。ハードウエアへのアクセス直前まで動作させ、ハードウエア入出力のみをキーボードやスクリーンに置き換えるようにします。プログラムは以下のように記述します。

```
#ifdef BCM2835
  :  実機を使うコード
#else
  :  シミュレーションコード
#endif
```

こうしておけば、普段はシミュレーション環境（実機を使わない環境）でプログラムを実行できます。BCM2835 を#define すれば、実機で走らせることができます。ハードウエア依存部分は、できるだけ小さくなるようにします。

5.4.2デバグ用コード

シミュレーション環境だろうと実機環境だろうと、動作を確認したくなることがあります。その時は、以下のように記述して、その時の変数をコンソールスクリーンに表示させるようにします。

```
#ifdef DEBUG
print x, y, …
#endif
```

シミュレーション環境やデバグで犯しやすいミスを指摘しておきます。先に定義した runpython というシェルスクリプトでは、スクリーンへの出力は問題ないのですが、キーボードから入力を与えることができません。整形された Python プログラムの標準入力がパイプになっているからです。この場合には、いったん cpp の出力をファイルに入れて、それを実行させます。前のクリーンアップの節を見てください。

5.4.3検証用代替プログラム（スタブ）

ソフトウエアの検証をするとき、別モジュールの動作を模擬する代替モジュールをスタブといいます。スタブ（stub）には切り株とか、ちびた鉛筆というような意味があります。検証の主眼ではないが、それがないと目的とするモジュールの動作が確認できないときに使います。

今回のプロジェクトでは、モジュール毎に評価用スタブ（test_XXX.py）を用意して、広い検証範囲を確保するようにしました。そのため、モジュールを単体で動作させるためにファイル末尾によく見られる、if __name__ ==’__main__’: 以下の部分は用意していません。

6　システム構成の検討

この章と次の章では、フェーズ 3 の開発となる、プロトタイプの設計と検証を行います。採用したアプローチが実用的かどうかという評価と改善策の検討は、その次の章で行います。

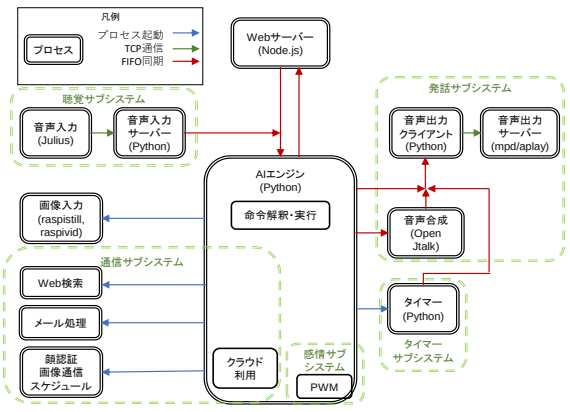
この章では、（少なくとも）実証モデルまでのシステム構成（ソフトウエアアーキテクチャ）を設計します。また、既存ソフトウエアの処理を高速化する工夫も考えます。

6.1　マルチプロセスシステム

秘書ロボットの作業にはさまざまな種類があり、ときには処理を並行させて応答を改善する必要があります。つまり複数の処理プログラムが（人間の時間感覚で見て）同時に走るシステムです。同じプログラムコードが、データのみ変わって複数実行されることもあります。それぞれの実行単位（プログラム＋データ）をプロセスと（ときにタスクとも）呼びます。プロセスは、OS の制御のもと、細切れに実行されるので、複数のプロセスが並行して走っているように見えるのです。

こう考えると、サブシステム＝プロセスと考えやすいのですが、サブシステムが複数のプロセスで構成される場合もあります。また、以前に複数のサブシステムとして仕様にまとめたものが、ひとつのプロセスとして実装されることもあります。独立性の高いプログラムモジュールを順番に実行すればいい場合には、同じプロセスにします。

以上を考慮して、前に設計したサブシステム構成をマルチプロセスシステムに書き直したものが下の図です。



マルチプロセスシステム

図中で黒い二重囲いがプロセス、緑の二重囲いがサブシステムです。赤と緑の矢印が、プロセス間の通信（データのやり取り）を表しています。

個々のプロセスは、必要なデータを受け取ってから処理を行います。このときデータ届いていないと、そこで待ち状態（プロセスの実行が止まる：ブロックするという）になり、データが届けば処理を再開します。

一方青い矢印は、新しいプロセスを生成（必要なメモリを割り振って実行させること）することを表しています。生成させる側の「親プロセス」は、「子プロセス」を生成したら、すぐに処理を再開します。

こうやって将棋倒しのように複数のプロセスが実行されていきます。同時に OS は、定期的に実行中のプロセスを待機（休止）させ、待機中の他のプロセスを実行させることもします。こうした時分割と待ち合わせ同期で、複数のプロセスが（見かけ上）同時に実行されていくのです。

発話サブシステムの音声出力と、聴覚サブシステムをそれぞれ二つのプロセスに分解しているのは、一見冗長に見えます。これは既存ソフトウエアの入出力をロボットの入出力に繋ぐために必要な処置です。

6.2　プロセス間通信

プロセス間の待ち合わせと情報交換を実現する手段が、プロセス間通信です。送り手側のプロセスから、受け手側のプロセスに何らかのデータを送りつける機能です。データを送りつけられる回数が、受け取って処理する回数より多くなってくると、受け手側にデータが溜まっていきますが、送りつけた順番に処理できるようになっています。

このプロセス間通信を実現する方法は何通りもありますが、ここでは 3 種類だけを使っています。

6.2.1TCP/IP 通信

インターネットでおなじみの TCP/IP を使って、同一機器（Raspberry Pi ）上、あるいは別 PC 上のプロセスとの間で通信を行います。同一機器内で通信をするときは、IP アドレスにループバックアドレス（127.0.0.1:<ポート番号>）を指定します。

送信側と受信側で、それぞれ通信ソケットを作り、その間を接続します。各プロセスは、ソケットにデータを送ったり、取り出したりするだけで済みます。

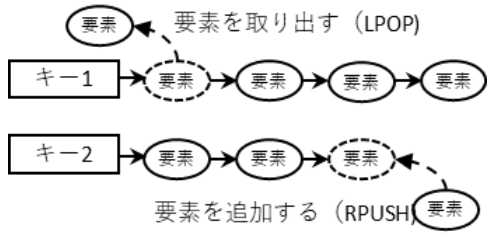
既存ソフトウエア Julius は、TCP/IP を使って認識結果を知らせるようになっています。そこで、音声入力サーバーはこれを受信して AI エンジンに与えるように設計します。

ただし TCP/IP では、クライアント側が通信要求を行うまでに、サーバー側のプロセスが立ち上がって、要求を受け付けられるようになっている必要があります。そうでないと、「接続に失敗」するエラーが起きてしまいます（リトライしたり、積極的に利用したりもする）。

6.2.2FIFO 通信

秘書ロボットのプロセス間で送りつけられるデータは比較的単純で短いから、もっと簡単で高速な手段を多用することにします。

温度コントローラで採用した、メモリ上の Redis データベースを FIFO（First-in-First-out）として使う方法です。



キー付きリストによる FIFO 実現

上の図のように、データベース上でキー（名前）ごとにリストを作り、右端から要素を追加（right-push）していき、取り出すときは左端から取り出（left-pop）します。取り出しはブロッキング型にします。空のリストから要素を取り出そう（BLPOP: blocking left-pop）とすると、その時点でプロセスが休止（ブロック）し、リストに新しい要素が付け加わったときに再開されます。

Redis サーバーはシステムと一緒に立ち上がるので、RPUSH する側が立ち上がっていなくても、BLPOP を行うことができる（データがないのでブロックしますが）ので、プロセスを立ち上げる手順や時間を考慮する必要がありません。実際の Redis サーバーへの要求は、TCP（ポート 6379）を通して行うのですが、Redis ライブラリを通すと、ずっと手軽に使えます。

Python や C で BLPOP をしようとしたとき、そのリストが空だと、プログラムはその場所で止まってしまいます。そのため、受信はプログラム上の一か所に限定されます（二つ以上のキーを与えて、データのある方から取り出すことはできる）。前ページのマルチプロセスシステムの図で、赤い矢印の終点は常に一つであることに注意してください。

いっぽう JavaScript では、BLPOP の処理は要求を登録するだけで、次の命令に進んでいきます。受信があると割り込み処理が実行されるので、複数の FIFO を同時に待ち受けることができます。

必要になるキーの名前を定義しておきます。実際に使うキーは別の（redis.h で定義する）テキストですが、設計時は名前だけで足ります。送信するプロセスが複数の場合があります。

キーの名前	送信プロセス	受信プロセス
REDIS_TO_AI	音声入力サーバー Web サーバー	AI エンジン
REDIS_TO_WEB1 REDIS_TO_WEB2	AI エンジン タイマー	Web サーバー
REDOS_TO_OPENJ	AI エンジン タイマー	音声合成
REDIS_TO_MPC	AI エンジン 音声合成 タイマー	音声出力クライアント

Redis キーの名前（プロトタイプ用のみ）

6.2.3シグナル

OS を介して、相手に割り込みを起こさせます。Ctrl+C キーを押して、プログラムを終了させるときに使っている方法です。この方法は実証モデルから使用するので、その時に説明します。

6.3　定型音声ファイル

音声合成は、それなりに CPU パワーを消費するので、決まり文句は事前に音声ファイルにしておきます。これを発声している間に、時間のかかる検索などを開始しておけば、処理時間を短く感じさせる効果もあります。

Web に表示するテキストから、音声合成で.wav ファイルを作っておきます。これは、Raspberry Pi ZERO より CPU パワーがある PC で作ると効率的です。もちろん Raspberry Pi でも行えます。何回も使うので、シェルスクリプトを作っておきましょう。/home/chuji/の個所は、それぞれのユーザーディレクトリに変更して使います。作業は音声ディレクトリ /home/chuji/voice で行います。

この後で何度も出てきますが、黄色い欄にファイル名を、白い欄にファイルの内容を表すことにします。

create_voice

```
cat $1.txt | open_jtalk -m
/home/chuji/htsvoice/takumi_sad.htsvoice -x
/var/lib/mecab/dic/open_jtalk_dic_utf_8-1.11 -
ow ./$1.wav
```

このシェルスクリプトを実行可能（chmod +x）にし、テキストファイル（.txt）から音声ファイル（.wav）を作ります。

```
$ chmod a+x create_voice

$ ./create_voice hello
```

下表のようなテキストファイルと定型音声ファイルを用意しました。テキストファイルと音声ファイルの名前は、拡張子以外は同じです。

音声ファイル名(.wav)	テキスト(.txt)
ah	（アー）
ahah	（アッアー）
yes	はい
hello	こんにちは
good_morning	おはようございます
shutdown	システムをシャットダウンします
good_night	おやすみなさい
bye	さようなら
today	きょうは
is	です
now	いま
weather	今日の天気は
timer_start(_○)	（○番）タイマーを起動しました
timer_expired(_○)	（○番）タイマーの時間が来ました
scheduled_time	予定時間になりました
who_are_you	どなたですか
your_name	名前を教えてください
just_name	カオナシも、「です」も要りません
you_look_happy	うれしそうですね
you_look_angry	怒っていませんか
you_look_concerned	なにか心配事でもありますか
you_look_sad	なんだか悲しそうですね
wait_a_moment	ちょっと待ってください
Google_will_tell	グーグルに応えさせます

定型音声ファイル（プロトタイプ用のみ）

このうち ah と ahah は、返事をする前と、命令を求めるときに発声させます。「千と千尋の神隠し」でカオナシが出す声です。音声合成は行わず、声優をつとめた中村彰男さんの音声をビデオから抜き出しました。これも著作物なのですが、ファイルは配布せず、私的な用途に限定することで許してもらいます。

6.4 音声コマンド辞書

Julius の認識精度を上げるため、秘書ロボットで使われる音声コマンドを辞書として用意します。

最低限必要なものとしては、カオナシ特有のコマンドを語彙（ボキャブラリ）集にします。この語彙は、そのままコマンドの解釈にも使えます。

発見したい語句	発音（かな書き）	発音表記
カオナシ	かおなし	k a o n a sh i
こんにちは	こんにちは	k o N n i ch i w a
シャットダウン	しゃっとだうん	sh a q t o d a u N
さようなら	さようなら	s a y o: n a r a
さようなら	さよなら	s a y o n a r a
今日	きょう	ky o:
何日	なんにち	n a N n i ch i
今何時	いまなんじ	i m a n a N j i
天気	てんき	t e N k i
予定	よてい	y o t e i
調べて	しらべて	sh i r a b e t e
時間	じかん	j i k a N
分	ふん	f u N
分	ぷん	p u N
経ったら	たったら	t a q t a r a
教えて	おしえて	o sh i e t e
知らせて	しらせて	sh i r a s e t e
私を覚えて	わたしをおぼえて	w a t a sh i o o b o e t e

カオナシ専用語彙集

これ以外に、数字の読みが必要です。たとえば「十」は「じゅう」、「とう」、「じゅっ」などと読みますが、標準辞書に掲載されているので省略しました。

発音表記はヘボン式で、子音と母音の間にスペースが入ります。「ん」は'N'、長音は':'、「っ」は'q'で表記します。発音表記が分かりにくい方は、ひらがなから自動生成する方法が金丸隆志さんの説明（https://raspibb2.blogspot.com/2017/03/raspberry-pi-julius-lirc.html）に従って、自動変換してください。

ロボット用の辞書に書き直した結果を kaonashi.dic として用意します。Julius プロジェクトの grammer-

kit にある datetime.voca に含まれる語彙のうち必要そうなものに、上の語彙を追加したのがこのファイルです。ファイル名の拡張子には制限がなく、自由に選んでも構いません。先頭の二行は、言葉の前後を区切る無音区間を指定しています。このファイルは長いので、この本に掲載していません。プロジェクトファイルを参照してください。

設定ファイル dictation-kit-4.5/main.jconf を修正して、辞書ファイルの指定を変更します。246 行目の指定を#でコメントアウト（青字）し、その後ろに新しい辞書を指定する行を追加（赤字）します。

Main.jconf（一部のみ）

```
:
:
#-v model/lang_m/bccwj.60k.htkdic
-v /home/chuji/dictionary/kaonashi.dic
:
```

元の辞書 bccwj.60k.htkdic は 64,000 行以上ありますが、kaonasi.dic は 145 行しかないので、処理時間の短縮が期待できます。「こんにちは」とか「カオナシ」と話しかけると、6〜7 秒で認識できました。その代わり、辞書にない言葉は正しく認識できません。決まった言葉だけを認識できればいいのなら、もう少し辞書を充実させれば対応できそうです。いっぽうインターネット検索をしようとすると、検索語を正しく認識しなければなりません。選択肢としては、

1. 実績のある大きな辞書にカオナシ特有の言葉を追加しする（認識にかかる時間は 15 秒程度）
2. カオナシ専用辞書を使う（認識できる範囲が限定されてしまう）
3. クラウドサービスを使う（使用料がかかる）

プロトタイプ（フェーズ 3）では、選択肢 1 で開発を進めます。評価の結果を見て、処理速度の改善を検討することにしました。

選択肢 1 を採用するため、bccwj.60k.htkdic と kaonasi.dic をよく比較した結果、後者にはあるが前者に含まれないのは、「カオナシ」と「何時（なんじ）」だけであることが分かりました（「何時（いつ）」は載っていました）。そこで、この二語を bccwj.60k.htkdic の 10023 行目の後ろと、22049 行目の後ろに追加しました。@以下は、同じエントリーの中で現れる確率（その常用対数）を設定します。

bccwj.60k.htkdic（赤字の追加周辺のみ）

```
（省略）
```

```
:
カオ+名詞          [カオ]  k a o
カオナシ+感動詞 [カオナシ]      k a o n a sh i
カオス+名詞        [カオス]       k a o s u
カオリン+名詞     [カオリン]      k a o r i N
カオル+名詞        [カオル]       k a o r u
:
（中略）
:
何方+代名詞       @-0.0193  ドチラ  [何方]  d o ch i r a
何方+代名詞       @-1.3617  ドナタ  [何方]  d o n a t a
何時+代名詞       @-0.0969  [何時]  n a N j i
何時+代名詞       @-0.6990  [何時]  i ts u
何時頃+名詞       [何時頃]       i ts u g o r o
何曜+名詞         [何曜]  n a N y o:
:
（省略）
```

Main.jconf の変更を元に戻し、デモモードで試してみると、たしかに「カオナシ、いま何時ですか」と認識できることが分かりました。認識精度を上げる方法は、もう少し調べてみようと思います。

コラム　LISP と Prolog

LISP（LISt Programming）は、FORTRAN に次ぐ古い「高級言語」です。1951 年、「AI」の名付け親でもある、ジョン・マッカーシーが設計し、当時の「人工知能」開発に使われました。データやプログラムはリストとして記述され、やたらとカッコが多いのが特徴です。Python の本で lambda（無名関数）なるものに出会って、「なんじゃい、これは？」と思った方もあると思います。実は、これは LISP の機能を受け継いでいる（文法は全く同じではない）のです。強力なリスト処理も併せて、「Python は LISP の後継」と言われることもあります。本文に出てきた ELIZA は、もともと LISP で書かれたプログラムですが、Python 等への移植例は、かなりあります。

Prolog（Programming in logic）は、定理を自動的に証明する研究から出てきた、論理を記述する言語です。「事実」と「規則」を記述できるので、LISP と「似ていないわけではない」くらいの類似性はあります。1980 年代に日本の通産省（現在の経産省）主導の国家プロジェクト『第五世代コンピュータ』計画では、（拡張して使う）基礎素材として採用されました。しかしこのプロジェクト、何の応用も実現できないまま、「当初の目的を達成した」として終了してしまいました。そのせいだけではないのですが、Prolog は教育用くらいしか使われなくなってしまいました。

6.5　人工知能の実装方法

AI の実装方法（どうやって実現するか）として、いまは機械学習がもてはやされています。これはニューラルネットワーク（神経回路網）という処理メカニズムの中に、神経素子間の結合の強さとして情報（知識）を蓄積していく手法です。「学習」という言葉が示すとおり、多くのデータを処理していく「経験」をとおして、知識を獲得できます。近年、大量のデータ入手と、処理速度の向上が可能になって、実用化が進むようになりました。クラウドサービスには、この技術をうまく使っているものがあります。しかし、個人の小規模システムでは、データ量も処理速度も、それに学習させる時間も、十分に確保できません。

それで、一世代前の手法である、エキスパートシステムを参考に考えます。これは、専門家（エキスパート）の知識を「規則」として記述し、現在の状況に対して「規則」をもとづく「推論」を行い、最適な結果を求めるアプローチです。分野によっては、人間の知恵を「それなり」に模倣できますが、当時は望みが高すぎたせいか、期待ほど普及しませんでした。

しかし秘書ロボットは「それなり」が目標なので、このアプローチでも十分です。設計の見本にしたのは、1966 年にワイゼンバウムが発表した ELIZA という心理カウンセラーを模倣するプログラムです。患者の言葉からキーワードを拾い出し、辞書にある応答（複数の応答があり、ランダムに選ぶ）を返すだけですが、「なんとなく」カウンセリングを受けている気になるという代物です。いわゆる「チャットボット」の原初版といったもので、専門家らしい知恵は何もないので、日本では人工頭脳ならぬ「人口無脳」とも呼ばれました。ただ秘書ロボットは業務に忠実であればいいので、毎回応答を変えるといった人間臭さは不要です。

プロトタイプでは、さらに簡略化して、規則と推論を、単純な if ～ then ～ else ～の文型で実現します。貧弱な CPU パワーで実装するためです。この方法では、規則が多くなるとプログラムが長くなるだけでなく、規則間の重複や矛盾を避ける記述が必要になり、面倒な実装になってしまいます。実現する業務が増えてきたら、規則集を辞書化するような、別のアプローチを検討することにします。

その場合でも、実際の業務を遂行するソフトウエアモジュールを用意し、AI の判断で呼び出すようにしておけば、大規模な改造にはなりません。

コラム　ELIZA と話す

今日でも電子無脳 ELIZA と話すことができます。emacs エディターをご使用の方は、M-x doctor（M-x は、ESC の次に x をタイプして入力します）で ELIZA との対話画面が出てきます。以下は会話の例です。「なんとなく」問診を受けているような……。

(Eliza) I am the psychotherapist. Please, describe your problems. Each time you are finished talking, type RET twice.

(Me) Hello, doc. My name is George. I have a problem.

(Eliza) Why do you say you have a problem?

(Me) Because I feel distressed since last week.

(Eliza) Is it because you feel distressed since last week that you came to me?

(Me) Yes, it is my problem.

(Eliza) Earlier you said you feel distressed since last week?

(Me) Yes, I did.

(Eliza) Maybe your life has something to do with this.

(Me) Eh... yes, it is one of my friends, whom I am worrying about.

(Eliza) What makes you believe that?

(Me) Because we had an argument last week.

(Eliza) Does it bother you that you had an argument last week?

(Me) Yes, very much.

(Eliza) You mentioned that you feel distressed since last week?

(Me) Yes, I did. I think the argument is the cause.

(Eliza) Perhaps your plans have something to do with this.

(Me) I don't have any plan at this moment.

(Eliza) I don't understand.

(Me) I mean... my problem is the relation to the friend.

(Eliza) Are you sure that you had an argument last week?

(Me) Yes, I am. We actually had one.

(Eliza) Is it because of the people you hang around with that you are going through all this?

(Me) Are you saying another friend of mine may be able to help?

(Eliza) I'll ask the questions, if you don't mind!

(Me) Oh, please do so. Go ahead.

(Eliza) Is it because of problems at school that you say do so go ahead?

(Me) Well... We were good friends when we were schoolkids. We are, since then.

(Eliza) Can you elaborate on that?

(Me) I like him and I think he does also. We are good friends.

：

7　プロトタイプの設計とサブシステムの検証

秘書ロボットの基本的な機能を実現させるプロトタイプ（フェーズ 3 開発）の設計に移ります。まずプロトタイプの目的を明らかにしておきましょう。

A. 秘書ロボットをマルチプロセスシステムとして構成する手法を習得する
B. プロセス間の同期・通信方法を確立し、サブシステムあるいはモジュールの検証にも利用する
C. 既存プログラムを（処理の詳細を知らなくても）サブシステムの一部として利用できるようにする
D. プロトタイプを評価することで、次のフェーズに移行する際の課題を明らかにする

この目的を達成するため、プロトタイプに実装する機能は最小限に抑え、開発・検証・評価を効率的に行えるようにします。秘書ロボットとしては、ほとんど使い物にならないことは承知のうえです。

次に、フェーズ 3 の開発対象となる業務をもう一度整理しておきましょう。

分類	機能	実現範囲
既存ソフトウエア	音声合成	プロセスのサーバー（常駐）化
	音声認識	プロセスのサーバー（常駐）化
付加機能	Web サーバー	画像表示を除いた大部分の機能
挨拶業務	業務開始	フル機能
	あいさつ	フル機能
	業務終了	フル機能
問い合わせ業務	日付照会	フル機能
	時刻照会	フル機能
事務作業業務	時間管理	フル機能

プロトタイプで実現する機能

この章の記述の順番を説明しておきます。最初に各サブシステムの概要設計を行います。サブシステムが複数のプロセスから成り立っている場合もあります。それぞれの機能仕様を定めますが、ステートマシン（後述）が必要になる場合は遷移表を作成しておきます。

次に各サブシステムの詳細設計と検証を、以下の順番（詳細設計部の節番号をつけています）で説明します。実際の設計はトップダウン（以下とは逆の順番）で行いますが、説明の分かりやすさと検証の容易さを重視しました。サブシステムを複数のプロセスや、さらにソフトウエアモジュールを使って構成する場合は、それぞれを設計・検証します。

7.2. 各部に共通するヘッダーファイル
7.3. 聴覚サブシステム
7.4. 発話サブシステム
7.5. タイマーサブシステム
7.6. Web サーバー
7.7. AI エンジン

この章では、各サブシステムやソフトウエアモジュールの検証までを説明し、秘書ロボットプロトタイプの総合的な動作確認と評価は次の章で行います。プログラムコード（インクルードファイルを含む）と検証プログラムは、検証用データの一部を除き、すべてこの章に掲載しています。

サブシステム間は Redis で通信しているので、Redis を介してデータを与え、結果は Redis を監視することで、サブシステムごとの検証ができるようになりました。

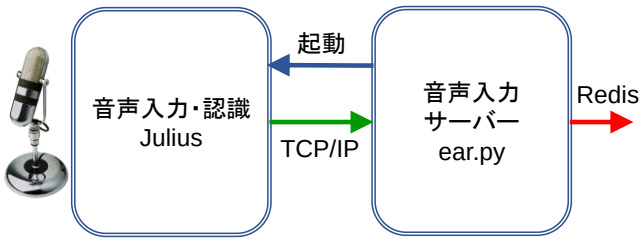
この章と次の章の作業は、とくに断らない限り、すべて kaonashi ディレクトリで行います。

7.1　サブシステムとプロセスの設計

前章で、秘書ロボットをマルチプロセスシステムとして構成しました。実現方法を考慮しながら、それぞれのサブシステムの仕様を詳細化してから、実際の開発に移ります。

7.1.1聴覚サブシステム

聴覚サブシステムの基本部分は、音声認識ソフトウエア Julius です。これに手を加えるのは辞書データだけにして、常駐させるようにします。認識結果を受け取り、AI エンジンに渡す音声入力サーバーもプロセスとして常駐させます。



プロセス	機能	言語
音声入力・認識	音声入力を日本語として認識する	C
音声入力サーバー	認識した日本語を AI エンジンに与える	Python

聴覚サブシステムの構成

Julius と音声入力サーバーの通信には、Julius 標準ポート番号 10500 を使います。

音声入力サーバー

音声入力サーバーは、Julius を起動し、認識した日本語を AI エンジンに渡す役割を果たします。

入出力	機能仕様
入力	Julius の認識した日本語（XML 文書）
出力	認識した日本語テキストを AI エンジンに与える

音声入力サーバーの入出力

音声入力サーバーは、Julius に対してはクライアントとして動作します。Julius からの結果待ちでブロックし、受信したデータから認識したテキストを取り出したら、Redis の REDIS_TO_AI キーに RPUSH します。

Julius からのメッセージは XML 形式で、以下のようなテキストが渡されます。ピリオドで始まる行が、一連の音声認識の最後を示しています。

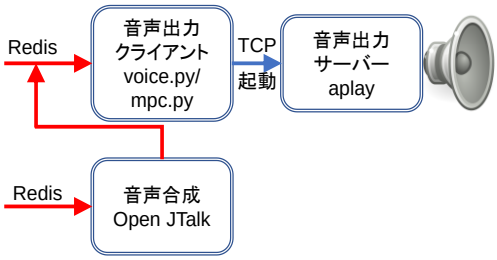
```
<STARTPROC/>
<INPUT STATUS="LISTEN" TIME="994675053"/>
<INPUT STATUS="STARTREC" TIME="994675055"/>
<STARTRECOG/>
<INPUT STATUS="ENDREC" TIME="994675059"/>
<GMM RESULT="adult" CMSCORE="1.000000"/>
<ENDRECOG/>
<INPUTPARAM FRAMES="382" MSEC="3820"/>
<RECOGOUT>
  <SHYPO RANK="1" SCORE="-6888.637695" GRAM="0">
    <WHYPO WORD="silB" CLASSID="39" PHONE="silB"
CM="1.000"/>
    <WHYPO WORD="上着" CLASSID="0" PHONE="u w a g i"
CM="1.000"/>
    <WHYPO WORD="を" CLASSID="35" PHONE="o"
CM="1.000"/>
    <WHYPO WORD="白" CLASSID="2" PHONE="sh i r o"
CM="0.988"/>
    <WHYPO WORD="に" CLASSID="37" PHONE="n i"
CM="1.000"/>
    <WHYPO WORD="して" CLASSID="27" PHONE="sh i t e"
CM="1.000"/>
    <WHYPO WORD="下さい" CLASSID="28" PHONE="k u d a
s a i" CM="1.000"/>
    <WHYPO WORD="silE" CLASSID="40" PHONE="silE"
CM="1.000"/>
  </SHYPO>
</RECOGOUT>
.
```

Julius の出力例

音声入力サーバーは、<WHYPO WORD=の後にある〝〟で囲まれたテキストを取り出して連結し、一連の日本語テキストとして AI エンジンに渡すことにします。

7.1.2発話サブシステム

発話サブシステムは、応答日本語テキストまたは音声ファイルをスピーカーから再生します。以下の 3 つのプロセスに分解して考えます。



プロセス	機能仕様	言語
音声合成	日本語テキストを音声に変換する	C
音声出力クライアント	音声ファイルを aplay に渡す	Python
音声出力サーバー	実際に音声を再生する	C

発話サブシステムの構成

音声出力サーバーとして利用できるものは各種あるのですが、発話のタイミング制御が面倒なので、プロトタイプでは最も簡単な方法として、ハードウエアの検証に使った aplay を、別プロセスとして毎回起動することにします。

音声合成

音声合成ソフトウエア Open JTalk をもとに、日本語テキストを音声ファイル（.wav）に変換します。

Open JTalk に以下の追加機能を組み込みます。

- 一回だけ処理するのではなく、Redis 入力待ちでブロックする、常駐プロセスに改造する
- 変換する日本語テキストを Redis（キー：REDIS_TO_OPENJ）から取り出す
- 変換した音声ファイルを Redis（キー：REDIS_TO__MPC）に登録する
- 音声ファイル名をプールして使いまわす

音声出力クライアント

音声出力クライアントは、Redis（キー：REDIS_TO_MPC）から再生要求を受け付けます。要求タイミングが異なる音声ファイルの再生を調整しながら、音声出力サーバーに再生タイミングを指示します。

入出力	機能仕様
入力	音声ファイル名と再生順番指定命令
出力	音声再生サーバーへの指令（aplay の起動）

音声出力クライアントの入出力

Redis から受け取るのは、命令＋音声ファイル名で、命令によって実行を調整します。

命令	動作
IMMEDIATE	音声ファイルを即時再生する
ADD	音声ファイルをプレイリストに追加する
PLAY (*)	プレイリストを再生する （音声ファイルのない IMMEDIATE と同じ）
WAIT (*)	JOIN 命令を受けるまで他の命令は実行しない
JOIN	音声ファイルをプレイリストに入れ、WAIT 状態だった命令を実行する

音声出力クライアントへの命令（*印にはファイルが付属しない）

WAIT と JOIN は、AI エンジンの発話要求の発声順番を維持するために設けました。音声合成に時間がかかるので、そのあとで発声する発話要求を待たせるための仕組みです。

その実現のため、音声出力クライアント自身が自分の「状態」を記憶し、状態毎に動作を定義していくことから始めます。このような仕組みを「ステートマシン」あるいは「有限オートマトン」と言います。ある状態（始状態）にあったとき、命令入力（イベント）に対する応答（どういうときに何をするか、次にどの状態にいるか）を指定することで、動作を定義していきます。以下のような状態遷移表にして整理すると、分かりやすくなります。状態ごとに、すべての命令入力に対して動作と終状態を決めてやります。

始状態	入力	条件	動作	終状態
状態名	入力名	○○のとき	○○をする	状態名

状態遷移表の記載項目

音声出力クライアントの場合は、状態は SENDING と WAITING の二つで、「条件」の記載は不要です。

始状態	受信命令	mpd コマンド/動作	終状態
-	（初期）	サーバーを初期化する	SENDING
SENDING	IMMEDIATE	音声を再生リストに加える リストを再生する	SENDING
SENDING	ADD	音声を再生リストに加える	SENDING
SENDING	PLAY	リストを再生する	SENDING
SENDING	WAIT		WAITING
SENDING	JOIN	音声を再生リストに加える	SENDING
WAITING	IMMEDIATE	命令をキューに入れる	WAITING
WAITING	ADD	命令をキューに入れる	WAITING
WAITING	PLAY	命令をキューに入れる	WAITING
WAITING	WAIT		WAITING
WAITING	JOIN	音声を再生リストに加える キューの命令をすべて実行	SENDING

音声出力クライアントの状態遷移表

音声出力クライアントは、ステートマシン voice.py と aplay を起動する mpc.py の二つのモジュールに分けて考えます。

7.1.3タイマーサブシステム

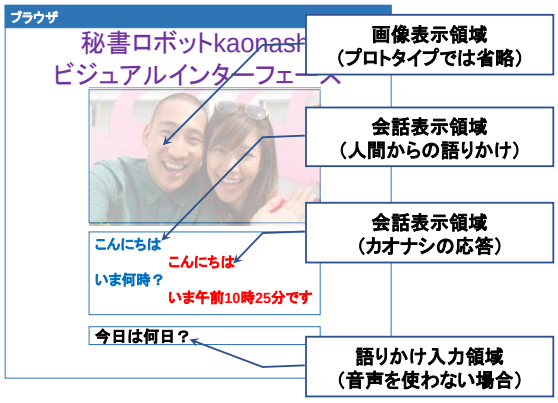
クッキングタイマーのように、「○○分後に教えて（知らせて）」と言えば、「タイマーを起動しました」と応え、○○分後に「タイマーの時間が来ました」と知らせるようにします。タイマーは一度にいくつも使えるようにし、「○番タイマー」と言わせることにします。起動時にタイマー番号と待ち時間を知らせるのは、AI エンジンの方で行うことにして、このサブシステムの機能は単純なものにします。

サブシステムは単一プロセスとし、AI エンジンから毎回起動させます。待ち時間だけスリープしたら終了を伝えるようにプログラムします。

7.1.4Web サーバー

Web サーバーは、カオナシとの会話を文字で表示するとともに、音声入力の代わりに、テキストで命令を与えられるようにします。これからは「カオナシ Web」と呼ぶことにします。

カメラなどからの画像を表示することを考えていますが、プロトタイプでは割愛しました。



Web 表示イメージ

サーバー(JavaScript)

Web サーバーは JavaScript で作成し、要求があったファイルをブラウザに送ります。ブラウザとの要求のやり取りは Socket.IO を通した通信で行います。

ブラウザからのテキスト入力は、Redis の REDIS_TO__AI キーに送ります。会話領域には直接表示しません。

Redis の REDIS_TO_WEB1 キー（命令側）と REDIS_TO_WEB 2 キー（応答側）に送られてきたテキストをブラウザに送ります。

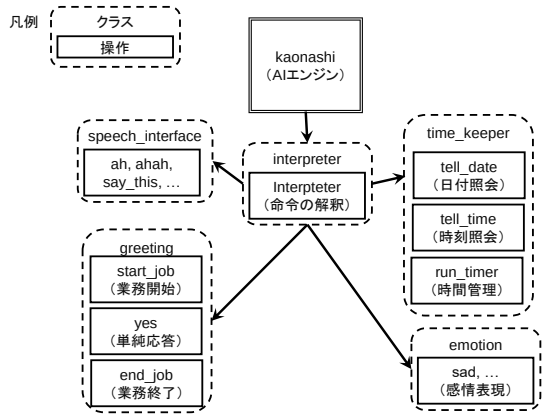
ブラウザ(HTML+JavaScript)

表示イメージを生成します。Socket.IO を通して、Web サーバーと通信し、テキスト入力を送信します。受信した会話テキストは、命令と応答の色を変えて表示します。プロトタイプでは、対話表示は各一行だけに簡略化しておき、履歴表示は次の段階にまわします。

設計を始めたとき、命令とロボットの応答は、図のような配置を考えていました。しかし LINE などの SNS では、自分の発言を右側に表示するのが慣習になっているので、このロボットでも真似をすることにしました。

7.1.5AI エンジン

AI エンジンは単一プロセスですが、複雑な処理がいくつもあるので、それぞれをオブジェクトとして設計します。結果は下図のとおりです。すべてのオブジェクト（点線で囲ったもの）は AI エンジン冒頭部 kaonashi が生成し、命令解釈実行部 interpreter から呼ばれます。



AI エンジンのオブジェクト設計

各オブジェクトの概要を表にまとめます。

クラス	操作	説明
kaonashi		AI エンジン冒頭部
interpreter		命令の解釈
	interprete	命令を解析する
speech_interface	ah, ahah	返事前後に発声する
	say, say_now	定型の返事をする
	say_this	発話と Web への表示
	speak_now	ここまでの音声を発声する
	synthesize	音声を合成する
	repeat	命令を Web に復唱する
greetings		挨拶
	start_job	業務開始の挨拶をする
	yes	単純な返事をする
	end_job	業務終了の挨拶をする
time_keeper		日付、時刻を知らせる
	tell_date	今日の日付を知らせる
	tell_time	現在の時刻を知らせる
	run_timer	タイマーを起動する
関数	get_time	テキストから時間を抽出する
emotion		感情を表現する
	sad 等多数	指定した感情を表現する

AI エンジンのオブジェクト

プロトタイプの会話は、一問一答式に限定されますが、命令者の息継ぎで命令が切れて検出される可能性があるので、解釈をする interpreter は簡単なステートマシンにしておきます。

出来上がったソフトウエアモジュールは、モジュール毎に単独で検証した後、サブシステムの評価を行います。

7.2　共通ヘッダーファイル

各ソフトウエアモジュールに固有なものを除き、共通して使えるヘッダーファイルを、ここにまとめておきます。ヘッダーファイルは、次節以降に出てくるものを含み、全て作業ディレクトリ　kaonashi の下にある include ディレクトリに置きます。

前にも説明したように、ヘッダーファイルのなかには、定義だけでなく、プログラム（cpp 処理で削除されない Python のコード）を含むものがあります。こういったヘッダーファイルは、すべて use-〇〇.h という名前にしています。

GPIO

Raspberry Pi の GPIO を使うための情報を gpio.h で定義します。実際には GPIO を直接操作することはないので、GPIO ピンの使途をまとめるための目的で作成しました。ここには掲載していないので、公開ファイルを参照してください。

システムコール

システムコール用の定義をいくつか用意します。まず subsystem 起動などの Linux コマンドを実行する

ための use-sys.h です。Python 用の宣言とコマンドが定義されています。

use-sys.h

```
/* システムコールの使用宣言
      初版： 2017/3/7 Chuji
      最新版： 2020/3/7 -- subprocess 追加
*/

#ifndef __USE_SYS

import sys
import os
import subprocess

#define SYS_SHUTDOWN  ['sudo', 'shutdown', '-h',
'now']
#define SYS_REBOOT  ['sudo', 'shutdown', '-r',
'now']

/* サブプロセスの起動 */
#define WORKING_DIRECTORY  r'/home/chuji/kaonashi'
#define sfork(x)  subprocess.Popen(x, shell = True,
cwd = WORKING_DIRECTORY)
#define fork(x)  subprocess.Popen(x, shell = False,
cwd = WORKING_DIRECTORY)

#define __USE_SYS

#endif
```

このインクルードファイルで重要なのは、subprocess.Popen を使って子プロセスを立ち上げるマクロ定義です。cwd は実行の作業ディレクトリを指定するためのものです。このプロジェクトで使うのは sfork と fork だけなので、subprocess の詳細は説明しません。詳しいことを知りたい人は、他の文献に当たってください。sfork はシェルに命令を解釈させるために使います。脆弱性に繋がりやすいと言われていますが、パイプやリダイレクトを含むシェルスクリプトを実行する必要がある場面のために用意しました。fork はコマンドと引数のみの場合に使います。

次は時間に関係する関数を呼ぶための定義ファイル use-time.h です。時刻によって挨拶を変えるためのしきい値も定義しています。

use-time.h

```
/* タイマー使用宣言
  初版：  2016/11/2 Chuji
  最新版：
*/

#ifndef __USE_TIME

from time import sleep
from time import time

#define MORNING_TIME  10
#define EVENING_TIME  17

#define __USE_TIME

#endif
```

パッケージライブラリ

数値演算パッケージを利用するための定義ファイル use-math.h です。

use-math.h

```
/* math ライブラリの使用宣言
      初版： 2016/11/3 Chuji
      最新版： 2016/11/3
*/

#ifndef __USE_MATH

import math

#define __USE_MATH

#endif
```

Redis

Redis FIFO を利用するための定義ファイル redis.h です。Python と JavaScript それに C 言語で使用します。

redis.h

```
/* REDIS データベースの定義
  初版： 2018/2/14 Chuji
  最新版： 2020/3/10 ... 言語依存性を排除

   言語依存性をなくすため、テキストは二重クオーテーションで囲
む
*/

#ifndef __REDIS


/* クライアント用 Redis インターフェース */
/* Python コード */
#define REDIS_SERVER  host = 'localhost'
#define REDIS_PORT  port = 6379

/* C 言語コード*/
#define REDIS_SERVER_C  "localhost"
#define REDIS_PORT_C  6379

/* データベース識別子 */
#define REDIS_FIFO  db=0 /* 内部通信用 FIFO */

/* Redis のコマンドオプション */
#define REDIS_ALL_ENTRIES  0 /* lrem コマンドですべてを
クリア */
#define REDIS_LAST_ENTRY  -1 /* リストの最後の要素を指
定 */

/* FIFO が空のときに読み込もうとするプログラムをブロック */
#define REDIS_NO_TIMEOUT  0

/* REDIS の BLPOP コール時に戻される配列の定義 */
#define REDIS_ID  0  /* キーID が戻される */
#define REDIS_DATA  1  /* FIFO から取り出されたデータ
*/

/* Redis キー（全言語共通） */
#define REDIS_TO_AI  "AI"
#define REDIS_TO_WEB1  "Web1"
#define REDIS_TO_WEB2  "Web2"
#define REDIS_TO_OPENJ  "OpenJTalk"
#define REDIS_TO_MPC  "mpc"

#ifndef REDIS_KEY
#define REDIS_KEY  REDIS_TO_AI
#endif
```

```
/* Redis API の取得 */

/* Python */
#define REDIS_API(x)
redis.StrictRedis(REDIS_SERVER, REDIS_PORT, x)
#define open_FIFO()  REDIS_API(REDIS_FIFO)

/* C 言語 */
#define open_FIFO_for_C()
redisConnect(REDIS_SERVER_C, REDIS_PORT_C)
#define blpop_c(c, key)    redisCommand(c,
"BLPOP %s 0", key)
#define rpush_c(c, key, x)  redisCommand(c,
"RPUSH %s %s", key, x)
#define FIFO_C      redisContext
#define FIFO_OUT    redisReply
#define FREE_RESOURCE(x)  freeReplyObject(x)
#define POPPED_STRING    element[1]->str

#define __REDIS

#endif
```

C 言語と共用する際に気をつけなければならないことがあります。C 用の redis ライブラリで使う定義と同じ名前を使ってはいけません。今回使った hiredis のヘッダーファイルを調べて、重複がないことを確認しました。

Python 専用の Redis 使用も定義しておきます。このファイルのなかで、redis.h も#include するようにして、プログラム中で#include するのは use-redis.h だけで良いようにします。

use-redis.h

```
/* use-redis.h
   初版: 2020/3/10 Chuji
   最新版:

Python から REDIS FIFO を使えるようにする

*/

#ifndef __USE_REDIS

#include "redis.h"

import redis

#define __USE_REDIS

#endif
```

TCP/IP

TCP/IP を介して Julius と通信するための定義ファイル tcp.h です。Python だけで使用します。

tcp.h

```
/* TCP 通信定義
  初版： 2020/3/5 Chuji

   プロセス間通信用    Python/C 共用
*/

#ifndef __TCP

/* IP アドレス:ポート */

#define TCP_LOOPBACK  '127.0.0.1'

#define TCP_PORT_JULIUS  10500
#define TCP_PORT_MPD  6600

#define TCP_JULIUS  (TCP_LOOPBACK, TCP_PORT_JULIUS)
#define TCP_JTALK  (TCP_LOOPBACK, TCP_PORT_MPD)

/* 受信バッファのサイズ */

#define TCP_BUFSIZ  2048

#define __TCP

#endif
```

コラム　日本発の大規模プロジェクト

日本が「世界に先駆けて」大規模開発プロジェクトを立ち上げたのは、第五世代コンピュータばかりではありません。1976 年に発足した『超 LSI 技術研究組合』では大容量メモリの製造技術を開発し、後に日本のドル箱産業とすることができました。この「成功体験の夢よ、もう一度」と願った官製プロジェクトが第五世代ですが、他にも例があります。

1985 年には、同じく通産省の『シグマ計画』が始まっています。こちらは「ソフトウエア生産の工業化」を目指していたのですが、全く成果が出ませんでした。残念ながら「失敗体験」を後の教訓にすることも繋がりませんでした。

官製プロジェクトではありませんが、同じころに東京大学の坂村健さんが産業界を糾合して始めた『トロン』プロジェクトは惜しいところまで行きました。独自のリアルアイム OS を作ろうというもので、組込み用の ITRON、ビジネス用の BTRON（今の Windows の位置づけ）、サーバー用の MTRON などの仕様を作り上げました。それをもとに各社が開発し、BTRON は教育現場への納入間近まで行ったものの、実現には至りませんでした。マイクロソフトを擁するアメリカの圧力だとも、国内企業間の争いのせいだとも言われていますが、世界最先端を行く開発構想だっただけに、全く残念なことです。いっぽう組込み用途の ITRON、特に小規模システム用の μITRON は市場で多く受け入れられています。

7.3　聴覚サブシステム

聴覚サブシステムは、音声認識ソフトウエア Julius と音声入力サーバーear の二つのプロセスからなります。

7.3.1Julius の常駐化

Julius はコマンドラインから、オプション-module を指定することで、（常駐プロセスとして）起動できます。音声入力ハードウエアを使用するので、実行は sudo を介して行います。

```
$ sudo julius -C <jconf ファイル> -module
```

そのためのシェルスクリプト run_julius を用意しました。どこからでも使えるよう、辞書ディレクトリはフルパスとして記述しています。

run_julius

```
sudo julius -C /home/chuji/dictation-kit-4.5/main.jconf -C /home/chuji/dictation-kit-4.5/am-gmm.jconf -module
```

誰でもこのシェルスクリプトを起動できるようにしておきます。検証のときだけでなく、音声入力サーバーが sfork()を介して Julius を起動するのにも使います。

```
$ chmod a+x run_julius
```

7.3.2音声入力サーバーの設計

音声入力サーバーは Julius とロボット AI を繋ぐ機能 ear.py と、XML の解析部 xml.py の二つのモジュールに分けました。すぐ後の Julius 単体検証で分かるように、XML 文書は切れ切れのバイト列として渡されるので、一行ごとに再構成してから解釈する必要があります。途中の状態を保持させるため、XML 解析部はオブジェクトとして設計します。

ヘッダーファイル

julius.h では、Julius の起動方法と XML 文法を定義します。

julius.h

```
/* julius.h
   初版: 2020/3/5 Chuji

  XML texts from Julius
*/

#ifndef __JULIUS

/* Julius 起動コマンド */
#define START_JULIUS  './run_julius'

#define WAIT_FOR_JULIUS_TO_READY  20 /* 立ち上がり時間 */
#define WAIT_FOR_JULIUS_TO_ANALYZE  0.01 /* 認識待ち時間 */

/* XML キーワード */

#define XML_START  '<RECOGOUT>'
#define XML_END    '</RECOGOUT>'
#define XML_DELIMITER  '.'

#define XML_TEXT  'WHYPO WORD'
#define XML_FROM  '\"'
#define XML_TO  '\"'

#define XML_START_VOICE  'silB'
#define XML_END_VOICE  'silE'

#define XML_NEWLINE  '\n'

/* テキスト定義 */

#define XML_FIRST_CHAR  0  /* テキストの最初 */

#define XML_EMPTY  ''  /* 空テキスト */
#define XML_DUMMY  ' '  /* 任意テキスト */
#define XML_PERIOD  '。'

#define XML_MARK  ':'  /* TCP ストリームの最初を示す */

#define __JULIUS

#endif
```

ear.py

メインモジュールは ear.py です。Julius を起動し、TCP/IP ポートから受信したデータを XML 解析部に渡します。XML 解析部で抽出された言語コマンドは、Redis を介して AI エンジンへ送信します。

マクロ定義#define でコードを切り分けています。

マクロ	#ifdef	#ifndef
HEARING_TEST	聞取り結果は表示のみ	聞取り結果を AI に渡す

ear.py のマクロ定義

ear.py では Julius を起動（sfork）したら、通信ができるようになるまで待ち、それから通信の接続を行います。Julius からデータ（の一部）を受け取ったら、xml 解析部に渡します。解析部が日本語を検出していたら AI エンジンに渡しています。

ear.py

```
/* 聴覚サブシステム：音声入力サーバー
    初版：2020/3/5 Chuji
  最新版：
*/

#include "include/tcp.h"
#include "include/use-sys.h"
#include "include/use-time.h"
#include "include/julius.h"

#ifndef HEARING_TEST

#include "include/use-redis.h"

fifo = open_FIFO()
```

```
#endif

#include "xml.py"

xml = xml_interpreter()

sfork(START_JULIUS) /* Julius の起動 */
sleep(WAIT_FOR_JULIUS_TO_READY) /* Julius の起動を待つ
*/

import socket       /* Julius との通信ポートを開く */
with socket.socket(socket.AF_INET,
socket.SOCK_STREAM) as julius:
  julius.connect(TCP_JULIUS)

  try:
    while True:
      /* ブロック型受信待ち */
      julius_out = julius.recv(TCP_BUFSIZ)

      xml.concatenate(julius_out)
      text = xml.get_japanese()

      if text != XML_EMPTY:

#ifdef HEARING_TEST     /* 日本語の認識テストをする場合
*/
        print('Julius recognized: ', text)
#else         /* AI エンジンと連携する場合 */
        fifo.rpush(REDIS_TO_AI, text)
#endif
      sleep(WAIT_FOR_JULIUS_TO_ANALYZE)
  except KeyboardInterrupt:
    pass
```

xml.py

次は XML 解析モジュール xml.py です。受信した XML テキストを一行ごとに分析し、音声認識した結果をテキストに返します。

オブジェクトの定義は以下のとおりです。

クラス	xml_interpreter	XML 解析部
属性	xml	XML 文書を順次追加して保持する
	text	解読した日本語を順次追加する
	japanese	解読が完了した日本語文
操作	concatenate()	テキストを XML 文書に追加する
	getline()	XML 文書から一行を取り出す
	parse()	XML 文書を解析する
	get_japanese()	解読した日本語文を取り出す

xml.py のオブジェクト

Julius から受信したバイト列は、操作 concatenate で一連の XML 文書として再構成されます。同時に解析を行います。操作 getline は、その XML 文書から一行を取り出します。操作 parse は、その一行の記述を解析し、音声認識した結果の日本語を再構成します。この動作は、日本語文を見つけるか、次の行がなくなるまで続けます。操作 get_japanese は、解析結果の日本語文を取り出します。

マクロ定義#define でコードを切り分け、動作の確認ができるようにしています。

マクロ	#ifdef	#ifndef
JULIUS_DEBUG	Julius から受け取った（部分）XML を表示する	表示しない
TEST_LINE	XML 文書一行を表示する	

xml.py のマクロ定義

xml.py のコードは以下のとおりです。

xml.py

```
/* xml.py  XML テキストを解析して、日本語テキストを探す
  初版： 2020/3/6 Chuji
  最新版：
*/

/* Object definition

Class xml_interpreter
属性:
  xml: XML 文書の一部
  text: 認識した日本語の一部
  japanese: 認識した日本語の全部
操作:
  concatenate: XML テ文書の一部を与える
  getline: XML 文書の一行を取り出す
  parse:  XML 文書を解析して、認識日本語テキストを再構成する
  get_japanese: 認識した日本語文を取り出す

*/

#include "include/julius.h"

class xml_interpreter:
  def __init__(self):
    self.xml = XML_EMPTY  /* 受信した XML 文書（一部）
*/
    self.text = XML_EMPTY  /* 認識した日本語の一部 */
    self.japanese = XML_EMPTY  /* 認識した日本語全文
*/

  /* 受信したバイト列をテキストとして保存する */
  def concatenate(self, octet_string):
#ifdef JULIUS_DEBUG
    print('$ ',octet_string)
#endif
    self.xml += octet_string.decode()
    self.parse()

  /* 保存したテキストから一行分を取り出す
     取り出せなかったら空テキストを返す */
  def getline(self):
    ret = XML_EMPTY
    pos = self.xml.find(XML_NEWLINE)
    if pos >= 0:  /* 改行があったら、そこまでを切り出す
*/
      ret = self.xml[:pos]
      self.xml = self.xml[pos + 1:]
    return ret

  /* XML 文書から一行ずつ解析し、認識テキストがあったら連結
していく
     先頭がピリオドの行があったら、解析を終了 */
  def parse(self):
    line = self.getline()
    while line != XML_EMPTY:
#ifdef TEST_LINE
      print (line)
#endif
      if line[XML_FIRST_CHAR] == XML_DELIMITER:
        self.japanese = self.text
        self.text = XML_EMPTY
      elif XML_TEXT in line:
        from_t = line.find(XML_FROM)
        len_t = line[from_t + 1:].find(XML_TO)
        fragment = line[from_t + 1: from_t + len_t
+ 1]
        if fragment == XML_END_VOICE:
            self.text += XML_PERIOD
        elif fragment != XML_START_VOICE:
            self.text += fragment
```

```
      line = self.getline()
    return

  /* 解析が終了した日本語文を返す。ないときは空テキストを返す
     日本語文を返したら、次の解析を始める */
  def get_japanese(self):
    ret = self.japanese
    if ret != XML_EMPTY:
      self.japanese = XML_EMPTY
      self.parse()
    return ret
```

7.3.3モジュールモード Julius 単体検証

検証用スタブ test_julius.py を用意します。あらかじめ Julius をコマンドラインから起動（run_julius）しておきます。test_julius.py は Julius との通信ポート（ソケット）を開き、データを受け取ります。データはバイト列として扱われるので、print 文では、UTF-8 にデコードして表示しています。

test_julius.py

```
/* 聴覚サブシステム：Julius 出力確認用スタブ
    初版：2020/3/5 Chuji */

#include "include/tcp.h"
#include "xml.py"

import socket

count = 1
with socket.socket(socket.AF_INET,
socket.SOCK_STREAM) as julius:
  julius.connect(TCP_JULIUS)

  try:
    while True:
      julius_out = julius.recv(TCP_BUFSIZ)
#ifdef GEN_XML
      print(XML_MARK, julius_out.decode())
#else
      print(count, XML_MARK, julius_out.decode())
      count = count + 1
#endif
  except KeyboardInterrupt:
    pass
```

この検証は、PC 上で SSH クライアント（以下では「SSH 窓」と表現します）を二つ立ち上げて行うと便利です。一方で Julius をモジュールモードで起動して（ready というメッセージが表示されて）から、他方で test_julius.py を起動すると、受信したデータが表示されます。

1:
```
$ sudo ./run_julius
 :
///////////////////////////////
///  Module mode ready
///  waiting client at 10500
///////////////////////////////
```

2:
```
$ cpp test_julius.py | python
1 : <STARTPROC/>
.

2 : <INPUT STATUS="LISTEN" TIME="1585536903"/>
.
```

```
3 : <INPUT STATUS="STARTREC" TIME="1585536905"/>
:
42 :     <WHYPO
43 :  WORD="こんにちは"
44 :  CLASSID="こんにちは+感動詞"
45 :  PHONE="
46 : k
47 :  o
48 :  N
49 :  n
50 :  i
51 :  ch
52 :  i w a" CM="0.344"/>
    <WHYPO WORD="。" CLASSID="&lt;/s&gt;"
PHONE="silE" CM="1.000"/>
  </SHYPO>
</RECOGOUT>
.
```

TCP/IP はストリーム型の通信で、読み込もうとするとそれまでに受信されたバイト列が返されます。一行ごとに結合させて、XML 文法に従って解読していく必要があることが分かります。

あとで検証に使うために、受信バイト列を保存しておきます。実行時に GEN_XML を#define しておくと、受信バイト列の冒頭に XML_MARK（実際には':'）とスペースをつけてプリントします。標準出力をファイル test_xml.txt（ファイル名は何でも構いません）にリダイレクトすれば、検証用の入力ファイルが作れます。

```
$ cpp -DGEN_XML test_julius.py | python
>test_xml.txt
```

XML_MARK をつけるのは、バイト列のなかにも改行記号が含まれているため、print 処理の最後に付け加えられてしまう改行を区別して見つけるためです。

7.3.4音声入力サーバー単体検証

音声入力サーバーは二つのモジュールからなります。後の検証で使うモジュールを先に検証するようにします。

XML 解析モジュール(xml.py)の検証

xml.py を検証するのに test_xml.py を用意しました。Julius から受信する代わりに標準入力から XML 文書を得るようにし、xml.py の解読結果を表示します。test_julius.py と test_xml.py は、ear.py を二つに分解して用意しました。いつもなら#ifdef で切り分けるようにするのですが、ファイル入出力がちょっと複雑なので、別々のファイルにしました。

test_xml.py で面倒だったのは、ファイルはテキストなのに、TCP/IP 通信で得られるバイト列として xml.py に与える必要があることと、test_julius.py が

付け加えた改行記号を削除することでした（あとで分かったことですが、test_julius.py のなかで `print(…, end='')` とすれば、改行記号は付け加えられません）。ファイルから読み取ったテキストを連結していくことで Julius から受け取ったデータ（バイト列）を再現し、先頭に XML_MARK があるテキストを見つけたら直前の改行記号を `[:len() -1]` で除去します。また、先頭の XML_MARK と空白文字も削除します。

EAR_TEST を#define していなければ認識した日本語テキストを表示します。#define してあれば、Redis FIFO に送信します。

test_xml.py

```
/* test_xml.py XML 解析の試験
*/

#include "xml.py"

#ifdef EAR_TEST

#include "include/redis.h"

import redis
fifo = open_FIFO()

#endif

xml = xml_interpreter()

julius_out = XML_DUMMY

with open('/dev/stdin') as f:

  for file_out in f:
    if file_out[XML_FIRST_CHAR] == XML_MARK:
      julius_out = julius_out[: len(julius_out) -
1]
      xml.concatenate(julius_out.encode())
      text = xml.get_japanese()
      if text != XML_EMPTY:
#ifndef EAR_TEST
        print('\nJulius recognized: ', text,
'\n')
#else
        fifo.rpush(REDIS_TO_AI, text)
#endif
      julius_out = file_out[2:]
    else:
      julius_out += file_out
```

XML 文書を手で入力するのは大変なので、Julius 単体検証のときに作成しておいた test_xml.txt というファイルを使います。このファイルは標準入力として与えるので、cpp の処理結果をいったん test.py（ファイル名は何でも構いません。このファイル名は、このあと何度も同じ目的で使います）というファイルにしてから実行します。

```
$ cpp test_xml.py >test.py; python test.py
<./test_xml.txt
```

test_xml.txt に含まれる日本語を目で探し、それが日本語文として表示されることを確認します。

音声入力サーバーモジュール(ear.py)の検証

ear.py が、Julius から得た解釈結果を正しく xml.py に与えているかどうかを検証します。
JULIUS_DEBUG を#define して、Julius から何が渡ってきたか確認しながら行います。この段階では Redis に渡さず、表示だけにするため、HEARING_TEST を#define しておきます。

```
$ cpp -DJULIUS_DEBUG -DHEARING_TEST ear.py |sudo
python
```

XML 文書の断片と、認識日本語が表示されれば検証は終わりです。この時点では認識間違いは気にしないことにします。

7.3.5聴覚サブシステムの検証

音声認識した日本語は、Redis を介して AI エンジンに渡すことになっています。ここでは二つのプロセス間通信を含む検証をするので、SSH 窓を二つ開いて行います。まず、一つ目の窓で検証用プロセスを立ち上げます。、

```
1: $ cpp test_ear.py | python
```

test_ear.py は次のようなプログラムで、Redis FIFO で受信したテキストを表示するだけです。

test_ear.py

```
/* 聴覚サブシステム検証用スタブ
   初版：2020/3/7 Chuji
*/

#include "include/use-redis.h"

fifo = open_FIFO()

try:
  while True:
    tag, text = fifo.blpop(REDIS_TO_AI,
REDIS_NO_TIMEOUT)
    print('Julius recognized: ', text.decode())

except KeyboardInterrupt:
  pass
```

第二の SSH 窓から音声入力サーバーを起動します。まずは、XML 文書をファイルから取り出して検証します。

```
2: $ cpp -DEAR_TEST test_xml.py >test.py; python
test.py <./xml_pattern.txt
```

認識した日本語が第一の SSH 窓側に表示されるはずです。

次に Julius まで含めて検証します。第一の SSH 窓のプロセスが終了していたら、もう一度 test_ear.py

を立ち上げます。第二の SSH 窓で音声認識サブシステムを起動します。プロセス Julius が走っているときは、先に kill します。ear.py を起動し、しばらく待ってから、なにか話しかければ、どういう日本語に変換されたか、第一の SSH 窓で確認できます。

```
2:  $ cpp ear.py | sudo python
```

助走期間

Julius プロジェクトの報告書にもあるのですが、Julius を立ち上げた直後は認識に失敗することがあります。Julius 自身が動作パラメータを自動調整していくので、何回か試すと認識精度が上がっていきます。ちょっと忍耐が必要なんですね。

聴覚サブシステムの起動

聴覚サブシステムの起動シェルスクリプト run_ear を作成しておきます。

run_ear

```
sudo cpp /home/chuji/kaonashi/ear.py | python
/home/chuji/kaonashi/cleanfile.py | python
```

シェルスクリプトが誰にでも実行できるように設定します。

```
$ chmod a+x run_ear
```

C 言語版 test_ear

test_ear.py と同じことを C 言語で書き直してみます。これは Open JTalk を常駐型に改造するための練習を兼ねています。

C 言語用の Redis インターフェースである hiredis には、最初の接続と、Redis コマンド実行という二つの関数しかありません。Redis コマンドは、テキストとして redisCommand という関数に与えてやるようになっていますが、redis.h のなかで rpush と blpop を別々のマクロ名として定義することで、使いやすくしました。

また、コマンド実行関数の戻り値は、さまざまなコマンドに対応するため、複雑な構造体になっています（hiredis.h に右上のような定義がある）。blpop の場合は、取り出したテキストは配列の 2 番目要素として返ってきます（1 番目要素にはキーが返される）。この取り出し方も redis.h で定義してありま す。この構造体のメモリ領域は関数内で用意されるので、用がなくなったら解放して再利用できるようにしなければなりません。この事情は rpush でも同じです。Python のインターフェースと比べると、かなり面倒ですね。

hiredis.h（一部のみ抜粋。コメントは著者）

```
:
typedef struct redisReply {
    int type; /* 返す要素の型* */
    long long integer; /* 単精度整数を返す場所 */
    double dval; /* 倍精度整数を返す場所 */
    size_t len; /* テキストの長さ */
    char *str; /* テキストを返す場所 */
    size_t elements; /* 配列の要素数 */
    struct redisReply **element; /* 配列を返す場所 */
} redisReply;
:
```

C 言語でもう一つ面倒なのは、テキストの結合です。Python ではテキストの A の後ろにテキスト B を付け加えるのには、A+= B と書けば済みます。C 言語にはテキスト結合関数 strcat があって、strcat(A, B)と書けば同じことができるように見えます。しかし、メモリ領域を確保するために、最初にテキスト変数 A のサイズ（最大文字数+1）を定義しておく必要があります。もし strcat(A, B)の文字数が A の大きさを越えてしまうと、とんでもないところにデータを書き込んでしまう可能性があります。何回も結合を繰り返すと、途中結果の文字数が分からなくなってしまいかねません。A のサイズを十分大きく取っておくことに加え、後で設計する model_jtalk.c 以降では、sprintf で新しいテキストを作るようにしました。

test_ear.c

```
/* 聴覚サブシステム検証用スタブ（C 言語版）
    初版：2020/3/12 Chuji
*/

#include <stdio.h>
#include <hiredis.h>
#include "include/redis.h"

int main(){

   FIFO_OUT *reply;

   FIFO_C *fifo = open_FIFO_for_C();
   printf("Start listening...\n");
   while(1){
      reply = blpop_c(fifo, REDIS_KEY);
      if((reply != NULL) && (reply->type ==
REDIS_REPLY_ARRAY))
         printf("%s\n", reply->POPPED_STRING);
      FREE_RESOURCE(reply);
   }
}
```

hiredis のインクルードファイルのありか（-I）と、hiredis ライブラリの使用（-l）をオプションとして指定して、先の第一の SSH 窓でコンパイル、起動してみます。

```
1:  $ cc -o test_ear test_ear.c - lhiredis
    -I/usr/local/include/hiredis
    :
    $ ./test_ear
```

前と同様に第二の SSH 窓で ear.py を走らせると、Julius が認識した日本語が第一の SSH 窓に表示されます。

Redis インターフェース検証用スタブ

ここまでやってきて、test_ear.py というスタブが、プロセスの検証に有効であることが見えてきました。プロセス間でデータを渡しあうのに Redis を使うことが多いので、test_ear.py を汎用化して、これからの検証に利用することを考えます。Redis キーをコマンドラインから与えるように test_ear.py を改造します。他の言語（Javascript や C 言語）が日本語文を rpush しても、Python が lpop するとバイト列だと認識してしまいます。それで、decode して日本語文だと解釈するように改造しました。

pop_redis.py

```
/* Redis からの POP データ表示
    初版：2020/3/29 Chuji
  最新版：
*/

#include "include/use-sys.h"
#include "include/use-redis.h"

args = sys.argv
key = args[1]

fifo = open_FIFO()

try:
  while True:
    tag, text = fifo.blpop(key, REDIS_NO_TIMEOUT)
#ifdef ASCII
    print('Text:', text, 'from:', key)
#else
    print('Text:', text.decode(), 'from:', key)
#endif

except KeyboardInterrupt:
  pass
```

このファイルを cpp で処理して、popper.py というファイルを作っておきます。popper.py の第一変数に Redis キーを与えればいいのですが、ここでは REDIS_TO_AI の代わりに、実際のキー値である'AI'を（引用符は付けずに）使います。次の操作は、test_ear.py の実行と同じ結果が得られます。

```
$ cpp pop_redis.py > popper.py
$ python popper.py AI
```

こんどは Redis にデータを送る方を用意します。枠組みはほとんど同じで、blpop の代わりに rpush しています。

push_redis.py

```
/* Redis へのプッシュ
    初版：2020/3/12 Chuji
  最新版：
*/

#include "include/use-sys.h"
#include "include/use-redis.h"

args = sys.argv
key = args[1]

fifo = open_FIFO()

try:
  while True:
    text = input()
    print('Pushed:', text)
    fifo.rpush(key, text)

except KeyboardInterrupt:
  pass
```

これを cpp で処理して、pusher.py というファイルにしておきます。両方のプログラムを同時に実行してみましょう。pusher.py に入力したテキスト'Hello'が、popper.py で受信できていることが分かります。

```
$ cpp push_redis.py > pusher.py
$  python popper.py AI & python pusher.py AI
:
Hello
Pushed: Hello
Text: Hello from: AI
World
Pushed: World
Text: World from: AI
```

この二つのスタブを使って、プロセスにデータを送ったり、プロセスの送ったデータを調べたりすることにします。

コラム　日本のロボット史（その 1）

日本におけるロボットの歴史をたどってみましょう。古くは平安時代から、お茶くみなどをする「からくり人形」がありました。1928（昭和 3）年には、「學天即」という空気圧方式の自動人形が作られたそうです。それ以後の歴史を箇条書きにしました。

1967　産業用ロボットが日本に初導入される
1970　早稲田大学が二足歩行ロボット WABOT 発表
1983　日本ロボット学会創立
1988　NHK 高専ロボコン開始
1997　第一回ロボカップ（ロボットサッカー選手権）
1999　犬型ロボット AIBO（ソニー）発売
2000　ホンダ二足歩行ロボット ASIMO 発表
2002　NHK 大学ロボコン開始
2006　手術ロボット da Vinci の 日本導入
2008　ロボットスーツ HAL のサイバーダイン社創設
2014　ペッパー（ソフトバンク）デビュー

7.4　発話サブシステム

発話サブシステムは、以下のプロセスに分解しています。

- 音声合成プロセス
- 音声再生クライアント
- 音声再生サーバー（既存なので設計不要）

設計とは順番を逆にし、音声再生クライアントを説明してから、音声合成に進みます。後者の検証に前者が必要だからです。

7.4.1音声再生クライアント

音声出力クライアントは、ステートマシン voice.py と aplay を起動する mpc.py のモジュールに分けてあります。mpc という名前は、Linux の音楽サーバーmpd（音楽再生デーモン）のクライアントである、mpc から取ったものです。

ヘッダーファイル

音声再生クライアントで使用する定義をインクルードファイル mpc.h で定義します。このファイルは C 言語でも使うので、直接 Python のコードを含めることはしません（定義部分には入れてもよい）。

mpc.h

```
/* mpd クライアントのインターフェース定義
  初版:  2020/3/10 Chuji
  最新版:

Python と C 言語共用なので、テキストは二重クオーテーションで囲む
*/

#ifndef __MPC

/* mpc コマンド */
#define MPC_IMMEDIATE  'I'
#define MPC_ADD    'A'
#define MPC_PLAY  'P'
#define MPC_WAIT  'W'
#define MPC_JOIN  'J'

/* 以下は Python 専用 */

/* mpc コマンド作成 */
#define mpc_directive(x,y)  x[0]+y
  /*  x:コマンド  y:音声ファイル  */

/* mpc コマンド分離 */
#define mpc_command(x)  x[0]
#define mpc_file(x)  x[1:]

/* mpd 用命令(aplay 版) */
#define MPD_COMMAND  ['sudo', 'aplay']

/* mpd 用命令(mpd 版) */
#define MPD_CONSUME_ON  1
#define MPD_RANDOM_OFF  0
#define MPD_REPEAT_OFF  0
#define MPD_SINGLE_OFF  0
#define MPD_CROSSFADE_OFF  0

/* mpc ステート */
#define MPC_SENDING  0
#define MPC_WAITING  1

/* mpc ローカル待ち行列 */
#define MPC_Q_NULL  ''
#define MPC_Q_EMPTY  []
#define MPC_Q_TOP  0
#define MPC_Q_REMAINS  1:


#define __MPC

#endif
```

ステートマシン voice.py

先に設計したステートマシンを組み上げます。ステートマシンはオブジェクト voice_client として実現します。

クラス	voice_client	ステートマシン
属性	state	ステートマシンの状態
	queue	WAIT 中に使う、命令の待ち行列
	mpd	mpd_client
操作	put_queue	命令を待ち行列の最後に入れる
	get_queue	待ち行列の先頭から命令を取り出す
	interprete	命令を解釈・実行する

voice.py のオブジェクト

デバグ用の表示を可能にするマクロ定義を用いています。

マクロ	#ifdef	#ifndef
DEBUG	受信命令を表示する	受信命令を表示しない

voice.py のマクロ定義

ステートマシンの動作が複雑になるのは、JOIN 命令を受けたとき、WAIT している間に待ち行列（キュー）に入っていた命令をまとめて処理するときです。そのなかに、ふたたび WAIT 命令があると、そこで待ち行列の取り出しを中止します。

voice.py

```
/* 音声出力クライアント
     初版： 2020/3/10 Chuji
     最新版：

Redis(REDIS_TO_MPC)からコマンドとファイル名を受け取って再生する

オブジェクト：voice_client

属性：
  mpd  mpd クライアント
  state  ステート
  queue  待機時に命令を保持する待ち行列

操作：
  put_queue  命令を待ち行列の最後に入れる
  get_queue  待ち行列の先頭から命令を取り出す
  interprete  命令を解釈・実行する

*/
```

```
#include "include/tcp.h"
#include "include/use-redis.h"
#include "include/use-time.h"

#include "mpc.py"


/* クライアントステートマシン */

class voice_client:
  def __init__(self):
    self.queue = MPC_Q_EMPTY
    self.mpd = mpd_client()
    self.state = MPC_SENDING

  def put_queue(self, string):
    self.queue.append(string)

  def get_queue(self):
    if not self.queue:  /* 空の場合 */
      return MPC_Q_NULL
    return self.queue.pop(MPC_Q_TOP)

  def interprete(self, request):
    directive = mpc_command(request)
    music = mpc_file(request)
#ifdef DEBUG
    print (self.state, directive, music)
#endif
    if self.state == MPC_SENDING:
      if directive == MPC_IMMEDIATE:
        self.mpd.add(music)
        self.mpd.play()
      if directive == MPC_ADD:
        self.mpd.add(music)
      if directive == MPC_PLAY:
        self.mpd.play()
      if directive == MPC_WAIT:
        self.state = MPC_WAITING
      if directive == MPC_JOIN:
        self.mpd.add(music)

    else:  /* mpc_state = MPC_WAITING */
      if directive == MPC_JOIN:
        self.mpd.add(music)
        self.state = MPC_SENDING
        req = self.get_queue()  /* 待ち行列のフラッシュ */
#ifdef DEBUG
        print('Retrieved:',req)
#endif
        while req != MPC_Q_NULL:
          /* 再び待ち要求があったら、フラッシュを停止 */
          if mpc_command(req) == MPC_WAIT:
            self.state = MPC_WAITING
            break
          self.interprete(req)
          req = self.get_queue()
#ifdef DEBUG
          print('Retrieved:', req)
#endif
      else:
        self.put_queue(request)  /* 待ち行列に入れる */
#ifdef DEBUG
        print('Pushed:', request)
#endif


/* 音声出力クライアント本体 */

/* 初期化 */
voice = voice_client()
fifo = open_FIFO()

try:
  while True:  /* FIFO からの要求を待って処理 */
    tag, req = fifo.blpop(REDIS_TO_MPC)
    request = req.decode()
#ifdef DEBUG
    print('Received:', request)
#endif
    voice.interprete(request)

except KeyboardInterrupt:
  pass
```

<u>再生サーバーのクライアント mpc.py</u>

オブジェクトの定義は以下のとおりです。

クラス	mpd_client	mpd 操作部
属性	client	mpd クライアント
操作	add	プレイリストにファイルを加える
	play	プレイリストを再生する

mpc.py のオブジェクト

mpc.py の機能は aplay を起動することです。aplay は再生ファイル名を何個でも受け付けられるので、プレイリストは mpc.py のなかで持つように設計しました。

以下のマクロ定義を使用しています。

マクロ	#ifdef	#ifndef
SIMULATION	mpc コマンドを表示する	コマンドを子プロセスに渡す

mpc.py のマクロ定義

cpp 処理時に、マクロ定義 DEBUG は voice.py と共通で定義されます。

aplay の起動には fork ではなく、subprocess.run を使っています。fork では子プロセスが親プロセスと同時に実行されますが、subprocess.run では子プロセスが動作を終えるまで親プロセスの実行が止まります。aplay の実行中に次の再生コマンドが来ても、再生が終わるまで待たせるように、この方法を選びました。

mpc.py

```
/* mpd クライアント
     初版： 2020/3/10 Chuji
     最新版：

Music Player Daemon を操作する

クラス: mpd_client

属性:
  client   mpd クライアントオブジェクト
  command  aplay コマンド

操作:
  add   プレイリストに音声ファイルを追加する
  play  プレイリストを再生する

*/

#include "include/use-sys.h"
#include "include/mpc.h"
#include "include/AI_commands.h"

class mpd_client:
  def __init__(self):
    self.command = MPD_COMMAND


/* 音声の追加 */
  def add(self, voice_file):
```

```
    self.command.append(VOICE_DIRECTORY +
voice_file)

/* 音声の再生開始 */
  def play (self):
#ifdef SIMULATION
    print(self.command)
#else
    subprocess.run(self.command)
#endif
    self.command = MPD_COMMAND
```

mpc.py の検証

voice.py の検証に使うので、まず mpc.py を先に検証します。単独で検証するため、簡単なテストプログラム test_mpc.py を用意しました。操作が二つしかないので、試す範囲はあまり広くありません。

test_mpc.py

```
/* mpc.py テストプログラム
     初版： 2020/3/30 Chuji
     最新版：
*/

#include "mpc.py"

mpd = mpd_client()

mpd.add('ah.wav')
mpd.add('hello.wav')
mpd.add('ahah.wav')
mpd.play()
mpd.add('good_morning.wav')
mpd.add('shutdown.wav')
mpd.add('bye.wav')
mpd.play()
```

DEBUG を#define して実行すると、aplay を実行するコマンド（実際にはコマンドとパラメータからなるテキストの配列）を表示してくれます。音声ファイル名が、フルパスで指定されていることが確認できました。

```
$ cpp -DDEBUG test_mpc.py |python
['sudo', 'aplay', '/home/chuji/voice/ah.wav',
'/home/chuji/voice/hello.wav',
'/home/chuji/voice/ahah.wav']

['sudo', 'aplay',
'/home/chuji/voice/good_morning.wav',
'/home/chuji/voice/shutdown.wav',
'/home/chuji/voice/bye.wav']
```

次に DEBUG を#define せずに実行すると、6 つの音声ファイルを順番に再生します。

```
$ cpp test_mpc.py |python
```

ステートマシン voice.py モジュールの検証

ステートマシン voice.py の検証には、mpc.py の機能を利用します。DEBUG を#define したら、音声再生クライアントとして実行します。別の SSH 窓で pusher.py を実行し、命令を与えていきます。

1:
```
$ cpp _DDEBUG -DSIMULATION voice.py | python
Received: Aah.wav
0 A ah.wav
Received: Ahello.wav
0 A hello.wav
Received: P
0 P
['sudo', 'aplay', '/home/chuji/voice/ah.wav',
'/home/chuji/voice/hello.wav']
Received: W
0 W
Received: Agood_night.wav
1 A good_night.wav
Pushed: Agood_night.wav
Received: Jbye.wav
1 J bye.wav
Retrieved: Agood_night.wav
0 A good_night.wav
Retrieved:
Received: P
0 P
['sudo', 'aplay',
'/home/chuji/voice/bye.wav',
'/home/chuji/voice/good_night.wav']
```

2:
```
$ python pusher.py mpc
Aah.wav
Pushed: Aah.wav
Ahello.wav
Pushed: Ahello.wav
P
Pushed: P
W
Pushed: W
Agood_night.wav
Pushed: Agood_night.wav
Jbye.wav
Pushed: Jbye.wav
P
Pushed: P
```

SIMULATION を#define せずに test_mpc.py を実行すると、指定通りの順番で音声ファイルが再生されます。

音楽再生クライアントへの命令をいちいち手入力するのは面倒です。検証のカバーする範囲を広げ、再現性のある検証のため、命令をテキストファイル test_voice.txt にしておき、pusher.py の標準入力に与えてやるようにします。

test_voice.txt

```
Aprep-1.wav
Aprep-2.wav
P
Aprep-A.wav
W
Iprep-C.wav
Jsynth-B.wav
Aprep-a.wav
W
Aprep-c.wav
W
Aprep-e.wav
P
Jsynth-b.wav
Jsynth-d.wav
```

test_voice.txt では、合成音声ファイルとの同期をとるための WAIT 命令と JOIN 命令の組み合わせを、以下の 3 パターンで試しています。各パターンの終

わりでは音声の再生をしています。定型音声は prep-〇、合成音声は synth-〇というファイル名を使っています。SIMULATION を#define し、実際に音声を再生するわけではないので、どんなファイル名でも構いません。定型音声はすぐに命令が送られますが、合成音声は合成に時間がかかる状況を模擬しています。

定型音声-1 定型音声-2	定型音声-A 合成音声-B 定型音声-C	定型音声-a 合成音声-b 定型音声-c 合成音声-d 定型音声-e

ステートマシンの検証パターン

命令の受信と待ち行列の出し入れを表示させると煩雑になるので、DEBUG は#define しないで実行します。

```
1:  $ cpp -DSIMULATION voice.py |python
    ['sudo', 'aplay', '/home/chuji/voice/prep-
    1.wav', '/home/chuji/voice/prep-2.wav']
    ['sudo', 'aplay', '/home/chuji/voice/prep-
    A.wav', '/home/chuji/voice/synth-B.wav',
    '/home/chuji/voice/prep-C.wav']
    ['sudo', 'aplay', '/home/chuji/voice/prep-
    a.wav', '/home/chuji/voice/synth-b.wav',
    '/home/chuji/voice/prep-c.wav',
    '/home/chuji/voice/synth-d.wav',
    '/home/chuji/voice/prep-e.wav']
```

```
2:  $ python pusher.py mpc <test_voice.txt
    Pushed: Aprep-1.wav
    Pushed: Aprep-2.wav
    Pushed: P
    Pushed: Aprep-A.wav
    Pushed: W
    Pushed: Iprep-C.wav
    Pushed: Jsynth-B.wav
    Pushed: Aprep-a.wav
    Pushed: W
    Pushed: Aprep-c.wav
    Pushed: W
    Pushed: Aprep-e.wav
    Pushed: P
    Pushed: Jsynth-b.wav
    Pushed: Jsynth-d.wav
```

意図どおりの順番で再生するコマンドができていたら、音声再生クライアントの検証は終わりです。

7.4.2Open JTalk の常駐化

音声合成機能の実装には、次の二通りの方法があります。

a. Redis インターフェースとファイル名の処理を Python プログラムとして作成し、そこから fork で毎回 Open JTalk を起動する
b. Open JTalk の一部を改造し、構想どおりの常駐プロセスにするのに必要な機能を組み込む

実現は a.の方が簡単です。mpc.py が aplay を起動するのと同じやり方ができるからです。しかし、ここでは敢えて b.のやり方をとります。Open JTalk の起動と初期化に時間がかかるというのが第一の理由ですが、C 言語を忘れないように使っておきたいという目的もありました。Open Jtalk の開発者たちのプログラムを勝手に改造するので、改造する人には結果に対する全ての責任があります。

<u>改造試験コード</u>

まず、聴覚サブシステムの検証のところで作成（練習）した test_ear.c をもとに、追加機能部分のみを実現した、model_jtalk.c を設計します。

model_jtalk.c

```
/* Open Jtalk プロセス化キット（機能の実証用）
    初版：2020/3/14 Chuji
*/

#include <stdio.h>
#include <hiredis.h>
#include "/home/chuji/kaonashi/include/redis.h"
#include "/home/chuji/kaonashi/include/mpc.h"

char file_name[32], mpc_command[32];
int file_no = 1;  /* ファイル名には 1 から 5 までの番号をつける */


int main(){
  char *text;
  FIFO_OUT *rp;

  FIFO_C *fifo = open_FIFO_for_C();
#ifdef DEBUG
  printf("Start listening...\n");
#endif
  while(1){
    /* ファイル名と音声再生クライアントへのコマンド作成 */
    sprintf(file_name,
"/home/chuji/voice/voice%1d.wav", file_no);
    sprintf(mpc_command, "%cvoice%1d.wav",
MPC_JOIN, file_no);

    /* Redis FIFO 空のブロック型読み出し */
    rp = blpop_c(fifo, REDIS_TO_OPENJ);
    if((rp != NULL) && (rp->type ==
REDIS_REPLY_ARRAY)){
      text = rp->POPPED_STRING; /* 音声にする日本語テキスト */
      FREE_RESOURCE(rp);
#ifdef DEBUG
      printf("%s ==> %s\n", text, file_name);
#endif

      /* ここで Open JTalk の音声合成機能を呼び出す*/
      /* 日本語テキスト：text； 出力ファイル名：file_name */

      rp = rpush_c(fifo, REDIS_TO_MPC,
mpc_command);
      FREE_RESOURCE(rp);
    }
    file_no++; /* ファイル名の使いまわし */
    if (file_no > 5)  file_no = 1; /* 使いまわすファイル名は 5 個 */
  }
}
```

今までの方針と異なり、プログラム中に使いまわすファイル名の数やディレクトリなどを直接埋め込んでいます。理解を間違うことはないし、このプログラムの中でしか使わないからです。音声ファイルのディレクトリ/home/chuji/voice/は皆さんの環境に合わせてください。

処理の大筋は以下のとおりです。Redis の REDIS_TO_OPENJ キーから日本語テキストを読み出し、音声合成をした「つもり」になって、その出力ファイル名を Redis の REDIS_TO_MPC キーに送ります。末尾の番号のみ異なるファイル名を 5 個（voice1.wav～voice5.wav）用意し、順番に使っていきます。これにより、音声合成要求が連続してきても、再生中のファイルを壊してしまう心配がなくなります。ファイル名を使いまわすのは、音声ファイルが無限に増えていくのを防ぐためです。

```
$ cc -DDEBUG -o model_jtalk_ear model_jtalk.c
-lhiredis -I/usr/local/include/hiredis
```

コンパイルすると同時に、音声ファイルディレクトリに voice1.wav～voice5.wav を用意しておきます。他の既存メッセージ音声をコピーすれば十分です。

まず単体で検証してみましょう。Redis を介して入力（REDIS_TO_JTALK）と出力（REDIS_TO_MPC）を行っているので、スタブを使って動作を確認します。SSH 窓を三つ開き、pusher.py と popper.py を起動してから、model_jtalk を走らせます。pusher.py に入力を与えると、音声ファイル名が使いまわされているのが分かります。

```
1: $ python pusher.py OpenJTalk
こんにちは
Pushed: こんにちは
お元気ですか？
Pushed: お元気ですか？
お世話になりました。
Pushed: お世話になりました。
いいお天気ですね。
Pushed: いいお天気ですね。
また今度お話ししましょう。
Pushed: また今度お話ししましょう。
それではさようなら。
Pushed: それではさようなら。
```

```
2: $ python popper.py mpc
Text: b'Jvoice1.wav' from: mpc
Text: b'Jvoice2.wav' from: mpc
Text: b'Jvoice3.wav' from: mpc
Text: b'Jvoice4.wav' from: mpc
Text: b'Jvoice5.wav' from: mpc
Text: b'Jvoice1.wav' from: mpc
```

```
3: $ ./model_jtalk
Start listening...
こんにちは ==> /home/chuji/voice/voice1.wav
お元気ですか？ ==> /home/chuji/voice/voice2.wav
お世話になりました。 ==>
/home/chuji/voice/voice3.wav
いいお天気ですね。 ==>
/home/chuji/voice/voice4.wav
また今度お話ししましょう。 ==>
/home/chuji/voice/voice5.wav
それではさようなら。 ==>
/home/chuji/voice/voice1.wav
```

今度は、mpc.py まで繋いで検証します。SSH 窓を四つ開いておき、それぞれでプロセスを立ち上げます。model_jtalk.c は DEBUG を#define せずにコンパイルしてみましょう。第 4 の SSH 窓で起動します。

```
1: $ cpp voice.py | python
```

```
2: $ python pusher.py mpc
W
P
```

```
3: $ python pusher.py OpenJTalk
こんにちは
```

```
4: $ cc -o model_jtalk_ear model_jtalk.c     -
lhiredis -I/usr/local/include/hiredis
$ ./model_jtalk
```

第 1 の SSH 窓で音声再生クライアントを起動し、第 2 の SSH 窓から、MPC コマンド'W'を送ります。次に第 3 の SSH 窓から Open JTalk に適当なテキストを送ります。次に第 2 の SSH 窓から MPC コマンド'P'を送ると、音声ファイルが再生されるはずです。これで、音声合成「以外」の動作が実現できたことが確認できました。

最後にすべての窓で Ctrl-C を押せば、それぞれのプロセスを終了することができます。

Open Jtalk の改造

Open Jtalk を改造して使うのは、以下の 4 点です。

1. hiredis ライブラリを使用して、rpush と blpop を介した Redis へのアクセスができるようにする
2. 音声合成する日本語テキストを、標準入力ではなく、Redis から得るようにする
3. 合成した音声ファイル名を使いまわし、合成結果のファイル名を Redis から送る
4. Redis から日本語テキストを受け取るごとに音声合成を行う、常駐型サーバーにする

具体的な作業ディレクトリは/home/chuji/open_jtalk-1.11/bin です。ここに Open Jtalk の本体（入り口）のソースファイルである open_jtalk.c と、そのコンパイルを指示する Makefile や Makefile.in があります。これらのバックアップを取ってから、改造に入ります。同じ名前のファイルが他のディレクトリにもあるので、間違えないようにしてください。

1 番目の作業として、hiredis を使用するのに必要だった、インクルードファイルのありかと、hiredis ライブラリの指定を行います。改造試験コードではコンパイラのコマンドオプションとして指定していましたが、make を使うための工夫が必要です。直接 Makefile に手を入れても良いのですが、./configure

を動かしているので、その前段階である Makefile.in を改造することにしました。オプションとライブラリ指定部に、以下のように追記します。

近くの記述で@top_srcdir@などとあるのは、インクルードファイルの#define と同じで、ソースコードのありかを定義しているだけなので、恐れることはありません。ファイルのありかをフルパスで与えればいいのです。

Makefile.in（一部を抜粋。赤字部を追加する）

```
（省略）
AM_CPPFLAGS = -I @top_srcdir@/text2mecab \
   -I @top_srcdir@/mecab/src \
   -I @top_srcdir@/mecab2njd \
   -I @top_srcdir@/njd \
   -I @top_srcdir@/njd_set_pronunciation \
   -I @top_srcdir@/njd_set_digit \
   -I @top_srcdir@/njd_set_accent_phrase \
   -I @top_srcdir@/njd_set_accent_type \
   -I @top_srcdir@/njd_set_unvoiced_vowel \
   -I @top_srcdir@/njd_set_long_vowel \
   -I @top_srcdir@/njd2jpcommon \
   -I @top_srcdir@/jpcommon \
   -I /usr/local/include/hiredis \
   -I @HTS_ENGINE_HEADER_DIR@

open_jtalk_LDADD =
 @top_srcdir@/text2mecab/libtext2mecab.a \
 @top_srcdir@/mecab/src/libmecab.a \
 @top_srcdir@/mecab2njd/libmecab2njd.a \
 @top_srcdir@/njd/libnjd.a \
 @top_srcdir@/njd_set_pronunciation/libnjd_set_pron
unciation.a \
 @top_srcdir@/njd_set_digit/libnjd_set_digit.a \
  @top_srcdir@/njd_set_accent_phrase/libnjd_set_acce
nt_phrase.a \
  @top_srcdir@/njd_set_accent_type/libnjd_set_accent
_type.a \
  @top_srcdir@/njd_set_unvoiced_vowel/libnjd_set_unv
oiced_vowel.a \
  @top_srcdir@/njd_set_long_vowel/libnjd_set_long_vo
wel.a \
  @top_srcdir@/njd2jpcommon/libnjd2jpcommon.a \
  @top_srcdir@/jpcommon/libjpcommon.a \
  /usr/local/lib/libhiredis.a \
  @HTS_ENGINE_LIBRARY@ -lstdc++

open_jtalk_SOURCES = open_jtalk.c
all: all-am
（以下省略）
```

2 番目から 4 番目までの処理のため、open_jtalk.c に改造を加えます。このファイルの 280 行目あたりから int main()という関数が始まるので、処理を追ってみましょう。最初に日本語テキストを収納する変数 buff を定義し、各種ファイルへのポインタを初期化しています。日本語テキストを取り出すファイルは不要なのでコメントアウトします。次に辞書と声色ファイルの指定を探し、ファイルをロードします（ここには処理時間がかかっているかもしれないので、最初に一回だけ実行するようにします）。さらにオプション指定を調べますが、このうち出力となる音声ファイル（*wavfp）の指定で、ファイルを開く部分はコメントアウトします。音声合成自身の記述は短く、Open_JTalk_synthesis()という関数を呼んでいます。直前にある日本語テキストをファイルから読みとる処理を Redis からの取り出しに置き換え、変換後に出力ファイルを閉じて、そのファイル名を Redis に送ります。ファイル名の使いまわしを含め、先の改造試験コードを加えれば改造は終わりです。結果は以下のとおりです。

C 言語ではインデントに意味がありません。プログラムの構造を見やすくするために使われているだけなので、改造部分の構造が見えるような書き方をしています。

追記しているのは、不要なステートメントのコメントアウトと、model_jtalk.c コードの埋め込みです。model_jtalk.c に含まれていなかった追加は、Redis から受けとったテキストを変数 buff にコピーするのと、音声ファイルをオープンするところだけです。

open_jtalk.c（改造箇所のみ抜粋。赤字部を追加する）

```
（省略）
/* added by chuji 2020/3/31 */
#include "hiredis.h"
#include "/home/chuji/kaonashi/include/redis.h"
#include "/home/chuji/kaonashi/include/mpc.h"
/* end of addition */

int main(int argc, char **argv)
{
   size_t i;

   /* text */
   char buff[MAXBUFLEN];

/* added by chuji 2020/3/31 for output file name */
   char file_name[32], mpc_command[32], *text;
   int file_no = 1;
   FIFO_OUT *rp;
   FIFO_C *fifo = open_FIFO_for_C();
/* end of addition */

   /* Open JTalk */
   Open_JTalk open_jtalk;

   /* dictionary directory */
   char *dn_dict = NULL;

   /* HTS voice */
   char *fn_voice = NULL;

   /* input text file name */
/* deleted by chuji 2020/3/31 -- text file is not
used.
   FILE *txtfp = stdin;
   char *txtfn = NULL;
end of deletion */

（中略）
   /* get options */
   while (--argc) {
      if (**++argv == '-') {
         switch (*(*argv + 1)) {
         case 'o':
            switch (*(*argv + 2)) {
/* deleted by chuji 2020/3/31 -- output file shall
be opened later.
            case 'w':
               wavfp = fopen(*++argv, "wb");
               break;
end of deletion */

（中略）
          default:
           fprintf(stderr, "Error: Invalid option
'-%c'.\n", *(*argv + 1));
           exit(1);
         }
```

```
/* deleted by chuji 2020/3/31 -  text file not used
      } else {
         txtfn = *argv;
         txtfp = fopen(txtfn, "rt");
end of deletion */
      }
   }
/* added by chuji 2020/3/31 -- to make a memory-
resident server */
while(1){
  sprintf(file_name,
"/home/chuji/voice/voice%1d.wav", file_no);
  sprintf(mpc_command, "%cvoice%1d.wav", MPC_JOIN,
file_no);
  rp = blpop_c(fifo, REDIS_TO_OPENJ);
  if ((rp != NULL) &&(rp->type ==
REDIS_REPLY_ARRAY)){
   text = rp->POPPED_STRING;
   strcpy(buff, text);
   wavfp = fopen(file_name, "wb");
   FREE_RESOURCE(rp);
/* end of addition */

   /* synthesize */
/* deleted by chuji 2020/3/31 -- buff text is
obtained through Redis
   fgets(buff, MAXBUFLEN - 1, txtfp);
end of deletion */

   if (Open_JTalk_synthesis(&open_jtalk, buff,
wavfp, logfp) != TRUE) {
      fprintf(stderr, "Error: waveform cannot be
synthesized.\n");
      Open_JTalk_clear(&open_jtalk);
      exit(1);
   }
/* added by chuji 2020/3/31 -- send result through
Redis and make a loop */
   rp = rpush_c(fifo, REDIS_TO_MPC, mpc_command);
   FREE_RESOURCE(rp);
   if (wavfp != NULL) fclose(wavfp);

   file_no++;
   if (file_no > 5) file_no = 1;
  }
}
/* end of addition */

   /* free memory */
   Open_JTalk_clear(&open_jtalk);

   /* close files */
/* deleted by chuji 2020/3/31 --- text file is not
used/defined
   if (txtfn != NULL)
      fclose(txtfp);
   if (wavfp != NULL)
      fclose(wavfp);
end of deletiion */
   if (logfp != NULL)
      fclose(logfp);

   return 0;
}

OPEN_JTALK_C_END;
#endif                          /* !OPEN_JTALK_C */
```

Open Jtalk の作成者たちの著作権を尊重し、改造版の再配布は控えます。ファイルの冒頭にある著作権に関する注意書きをよく読んでください。

同じディレクトリで　open_jtalk.c のコンパイルを行い、でき上がった実行ファイルをインストールディレクトリにコピーします。他の必要なファイルには変更がないので、Open Jtalk のインストール時にコピーしたものがそのまま使えます。オリジナルの Open JTalk と区別するため、名前を変更しておきます。英語の resident は「居住者」とか「居住している」といった意味がありますが、コンピュータの世界では「常駐」という意味で使われます。

```
$ cd /home/chuji/open_jtalk-1.11
$ ./configure
$ cd bin
$ make
$ mv open_jtalk resident_open_jtalk
$ sudo cp resident_open_jtalk /usr/local/bin
```

常駐型 Open JTalk の検証

最初はスタブを使って、Redis インターフェースの動作を検証します。model_jtalk で行ったのと同じ手順です。Open JTalk を起動するときに、出力ファイルの指定（-ow）は不要です。

1:
```
$ python pusher.py OpenJTalk
こんにちは
Pushed: こんにちは
いい天気ですね
Pushed: いい天気ですね
```

2:
```
$ python popper.py mpc
Text: b'Jvoice1.wav' from: mpc
Text: b'Jvoice2.wav' from: mpc
```

3:
```
$ resident_open_jtalk -m
/home/chuji/open_jtalk-
1.11/htsvoice/takumi_sad.htsvoice -x
/var/lib/mecab/dic/open_jtalk_dic_utf_8-1.11/
```

この後、音声ファイル（voice1.wav と voice2.wav）を aplay で再生して、pusher.py から与えた日本語が発話されるか確認します。

7.4.3サブシステム検証

Julius と音声再生クライアントを起動して、サブシステムとして動作を検証します。

1:
```
$ python pusher.py mpc
Aah.wav
Pushed: Aah.wav
W
Pushed: W
P
Pushed: P
```

2:
```
$ python pusher.py OpenJTalk
こんにちは
Pushed: こんにちは
```

3:
```
$ resident_open_jtalk -m
/home/chuji/open_jtalk-
1.11/htsvoice/takumi_sad.htsvoice -x
/var/lib/mecab/dic/open_jtalk_dic_utf_8-1.11/
& (cpp voice.py |python)
```

音声合成クライアントに与えた命令に従って、音声ファイルが再生され、W 命令で Open JTalk の音声合成との待ち合わせが実現できることを確認します。ラフな測定ですが、音声合成にかかった時間は、短いもので 1～2 秒、長いときでも 3～4 秒くらいでした。

最後に、発話サブシステムとして、音声再生クライアントと常駐型 Open JTalk を起動するシェルスクリプトを用意します。自動起動することを念頭に、ファイルはフルパスで記述します。編集後に chmod a+x run_voice で実行できるようにしておきます。

run_voice

```
resident_open_jtalk -m /home/chuji/open_jtalk-1.11/htsvoice/takumi_sad.htsvoice -x /var/lib/mecab/dic/open_jtalk_dic_utf_8-1.11/ &
cpp /home/chuji/kaonashi/voice.py | python
```

7.5 タイマーサブシステム

タイマーサブシステムは、毎回プロセスとして起動し、タイマー番号と時間はコマンドラインから与えるようにします。

デバグ用のマクロ定義をしてあります。

マクロ	#ifdef	#ifndef
DEBUG	時間が来たら Web に表示するメッセージと音声再生サブシステムに渡す命令を表示する	時間が来たら Web に表示するメッセージと音声再生サブシステムに渡す命令を Redis に渡す

timer.py のマクロ定義

タイマーを起動するのに使う定義をインクルードファイル timer.h にまとめました。

timer.h

```
/* タイマープロセスで使用する定義
   初版：2020/3/1 Chuji
   最新版：
*/

#ifndef __TIMER

/* 使用可能なタイマー */

#define AVAIL_TIMERS  5
#define TIMERS  ['1', '2', '3', '4', '5']
#define TIMER_DEFAULT  '1'

/* タイマー設定範囲 */

#define TIMER_MIN  30
#define TIMER_MAX  60*60

#define TIMER_MIN_STRING  '30 秒'
#define TIMER_MAX_STRING  '1 時間'

/* 日本語メッセージ */

#define TIMER_START_MESSAGE  '番タイマーを起動しました。'
#define TIMER_EXPIRED_MESSAGE  '番タイマーの時間が来ました。'
#define TIMER_INVALID_TIMER  '指定したタイマー番号が不正なので、1 番を使います。'
#define TIMER_TOO_SHORT  '指定した時間が短すぎるので、30 秒に変更しました。'
#define TIMER_TOO_LONG  '指定した時間が長すぎるので、1 時間に変更しました。'

#define START1  'timer_start_1.wav'
#define START2  'timer_start_2.wav'
#define START3  'timer_start_3.wav'
#define START4  'timer_start_4.wav'
#define START5  'timer_start_5.wav'

#define EXPIRE1  'timer_expired_1.wav'
#define EXPIRE2  'timer_expired_2.wav'
#define EXPIRE3  'timer_expired_3.wav'
#define EXPIRE4  'timer_expired_4.wav'
#define EXPIRE5  'timer_expired_5.wav'

#define TIMER_STARTED  [START1, START2, START3, START4, START5]
#define TIMER_EXPIRED  [EXPIRE1, EXPIRE2, EXPIRE3, EXPIRE4, EXPIRE5]

/* タイマー起動命令 */
#define START_TIMER(x, y)  ['python', '/home/chuji/kaonashi/timer_process.py', x, y]

#define __TIMER

#endif
```

タイマープロセス timer.py は以下のとおりです。

timer.py

```
/* プロセス型タイマー機能
     初版： 2020/3/11 Chuji
     最新版：

指定したタイマーを起動して、時間が来たら知らせる

$ cpp timer.py >xtimer.py; python xtimer.py 1 60

*/

#include "include/AI_commands.h"
#include "include/timer.h"
#include "include/use-time.h"
#include "include/mpc.h"
#include "include/use-sys.h"
#include "include/use-redis.h"

voice = TIMER_EXPIRED

args = sys.argv

timer_id = args[1]  /* タイマー番号 */
if timer_id not in TIMERS:
  print(TIMER_INVALID_TIMER, file = sys.stderr)
  timer_id = TIMER_DEFAULT
timer = int(timer_id)

timer_set = args[2]  /* タイマーの待ち時間（秒） */
timer_in_sec = int(timer_set)

if (timer_in_sec < TIMER_MIN):
  print(TIMER_TOO_SHORT, file = sys.stderr)
  timer_in_sec = TIMER_MIN
if (timer_in_sec > TIMER_MAX):
  print(TIMER_TOO_LONG, file = sys.stderr)
  timer_in_sec = TIMER_MAX

sleep(timer_in_sec)

#ifdef DEBUG

print (timer_id + TIMER_EXPIRED_MESSAGE)
print (MPC_ADD + VOICE_DIRECTORY + voice[timer - 1])

#else

fifo = open_FIFO()
fifo.rpush(REDIS_TO_WEB2, timer_id + TIMER_EXPIRED_MESSAGE)
fifo.rpush(REDIS_TO_MPC, MPC_ADD + VOICE_AH)
fifo.rpush(REDIS_TO_MPC, MPC_ADD + voice[timer - 1])
fifo.rpush(REDIS_TO_MPC, MPC_IMMEDIATE + VOICE_AHAH)

#endif
```

このファイルを cpp で処理したものを、xtimer.py として保存しておき、タイマー番号と時間を与えて起動すれば、タイマーサブシステムとして使えます。

まず、検証のため、DEBUG を#define して、動作を確認します。タイマー番号が範囲外だと 1 番を使い、時間が 10 秒未満だと 10 秒に、1 時間以上だと 1 時間に変更されることを確認します。

```
$ cpp -DDEBUG timer.py >xtimer.py
$ python xtimer 1 30
1 番タイマーの、時間が来ました。
```

確認が済んだら、Redis で発話サブシステムに繋ぎます（DEBUG を#define しない）。

```
$ cpp timer.py >xtimer.py
```

別の SSH 窓で popper.py を起動し、REDIS_TO_MPC と REDIS_TO_WEB2 を監視しておきます。それから xtimer.py を起動してみます。

```
1:  $ python popper.py mpc & python popper.py
    Web2
    :
    Text: 5 番タイマーの時間が来ました。 from: Web2
    ('Text:', 'A/home/chuji/voice/ah.wav',
    'from:', 'mpc')
    ('Text:',
    'A/home/chuji/voice/timer_expired_5.wav',
    'from:', 'mpc')
    ('Text:', 'I/home/chuji/voice/ahah.wav',
    'from:', 'mpc')
2:  $ python xtimer.py 5 30
```

Web サーバーと音声再生クライアントにデータが伝わっていることが確認できます。

こんどは監視を REDIS_TO_WEB2 だけにし、同時に音声再生クライアントを起動してから、タイマーを起動してみます。DEBUG 時のテキストが、日本語として発話されれば検証は終了です。

```
1:  $ python popper.py Web2 & cpp voice.py
    |python
2:  $ python xtimer.py 3 30
```

7.6 Web サーバー

Web サーバー（カオナシ Web）はサーバーそのものと、ブラウザへの表示を制御する HML 文書からなります。

7.6.1共通インクルードファイル

Web サーバーは、ブラウザや AI エンジンと通信するので、通信の約束事などをインクルードファイルに定義しておき、Node.js に渡す前に cpp で置換することにしました。

Socket.IO や JavaScript で使用する定義をインクルードファイル web_if.h に記述しておきます。Web サーバーとブラウザの両方で使用できるようにしてあります。ふつうの Web サーバーのポート番号は 80 か 8080 ですが、カオナシ Web はユーザが選べる範囲から 50010 にしました。他の番号でも構いません。

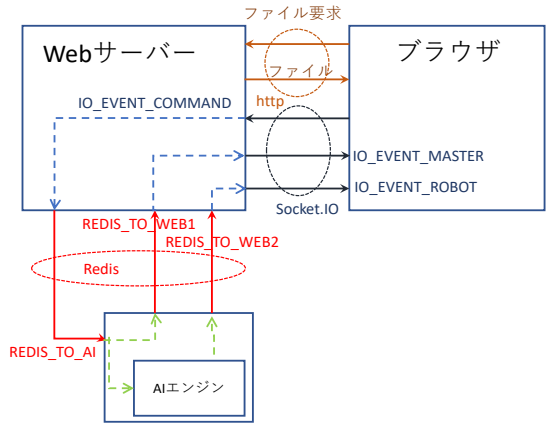
web_if.h

```
/* ロボットの Web インターフェース定義
  初版： 2020/3/19 Chuji
  最新版：
*/

#ifndef __WEB_IF

/* この定義は Web サーバーと html 用 Javascript のソースコード共用 */

/* Javascript、html 共用 SOCKET.IO 定義*/
/*  IP アドレスは実環境に合わせること */
#define URL_SOCKET_IO  'http://192.168.15.16:50010'
#define PORT_SOCKET_IO  50010

/* Javascript で記述した Web サーバー用 */

#define JS_FORK      'child_process'
#define JS_REQ_HTTP    'http'
#define JS_REQ_SOCKET_IO
'/usr/local/lib/node_modules/socket.io'
#define JS_REQ_FILE_SYSTEM  'fs'
#define JS_REQ_PATH    'path'
#define JS_REQ_REDIS
'/usr/local/lib/node_modules/redis'

#define JS_HTTP_STATUS_OK  200
#define JS_CHAR_UTF8    'utf-8'
#define JS_HTTP_HEADER(x)  {'Content-Type' : x}

#define JS_HTML_FILE
'/home/chuji/kaonashi/index.html'
#define NO_EXPLICIT_FILE  './'
#define CURRENT_DIR  '.'
#define INDEX_HTML  'index.html'

/* ファイル型定義 */

/* ファイル拡張子 */
#define EXT_HTML  '.html'
#define EXT_CSS    '.css'
#define EXT_JS    '.js'
#define EXT_PNG    '.png'
#define EXT_GIF    '.gif'
#define EXT_ICO    '.ico'
#define EXT_JPG    '.jpg'
#define EXT_JPEG  '.jpeg'
#define EXT_SVG    '.svg'
#define EXT_PDF    '.pdf'

/* ファイルの型 */
#define TYPE_HTML  'text/html'
#define TYPE_CSS  'text/css'
#define TYPE_JS    'text/javascript'
#define TYPE_PNG  'image/png'
#define TYPE_GIF  'image/gif'
#define TYPE_JPG  'image/jpeg'
#define TYPE_SVG  'svg+xml'
#define TYPE_PLAIN  'text/plain'
#define TYPE_PDF  'application/pdf'

/* socket.IO ライブラリへのポインタ */
#define SCRIPT_SOCKETIO  type="text/javascript"
src="/socket.io/socket.io.js"
```

```
/* Web サーバーと html に埋め込まれる Javascript 共用の
Socket.io イベント */

#define IO_EVENT_CONNECT  'connection'  /* サーバー
*/
#define IO_EVENT_CONNECTED  'connect'  /* クライアン
ト */
#define IO_EVENT_DISCONNECT  'disconnect'

#define IO_EVENT_COMMAND  'command'
#define IO_EVENT_ROBOT  'robot said'
#define IO_EVENT_MASTER  'master said'


/* html ページのデザイン */

#define WEB_TITLE_FONT  size = "4" color="red"
#define WEB_TITLE  秘書ロボット　カオナシ

#define WEB_ID_COMMAND  IO_EVENT_COMMAND
#define WEB_ID_ROBOT  IO_EVENT_ROBOT
#define WEB_ID_MASTER  IO_EVENT_MASTER

#define WEB_MESSAGE_RAW  "system message"
#define WEB_MESSAGE_RAW2  "Operation message"

#define __WEB_IF

#endif
```

7.6.2Web サーバー

Web サーバーの機能は次の図に示すように、ブラウザに要求されたファイルを渡すのと、Socket.IO のメッセージ付きイベントと Redis を中継することからなります。



Web サーバーの中継機能

Web サーバーの JavaScript コードは以下のとおりです。最初の部分ではヘッダーファイルをインクルードし、ライブラリの呼び出し窓口を作っています。関数 getType は、要求されたファイルの拡張子に応じて、http ヘッダーに入れるファイルの種類を作っています。関数 createServer はサーバー機能を立ち上げ、要求されたファイルを返しています。ファイルには HTML 文書やアイコン、スクリプトファイルなどがあります。

関数 create_message（create_event は定義しましたが、使っていません）では、要求があったときに Socket.IO を介してブラウザにテキストを送ります。

次の socket.on では Socket.IO からのイベントに対する処理を記述しています。
IO_EVENT_COMMAND を受信したら、付随しているメッセージ（ブラウザから入力した、ロボットへの命令）の冒頭にウェイクアップワード（「カオナシ」）を追加して、Redis 経由で AI エンジンに送ります。

Redis を介してメッセージ（ブラウザに表示するテキスト）を受信したら（show_master と show_robot）、Socket.IO 経由でブラウザに送っています。

web-if.js

```
/* 秘書ロボット用 Web サーバー
   オリジナル：温度コントローラ　2017/6/5-2019//7/10
   初版：2020/3/19
   最新版：
*/

#include "include/web_if.h"
#include "include/redis.h"
#include "include/AI_commands.h"

var http = require(JS_REQ_HTTP);
var socketio = require(JS_REQ_SOCKET_IO);
var fs = require(JS_REQ_FILE_SYSTEM);

/* ブラウザとのインターフェース */

/* 要求されたファイルの型認識 */
function getType(_url){
  var types = {
    EXT_HTML: TYPE_HTML,
    EXT_CSS: TYPE_CSS,
    EXT_JS: TYPE_JS,
    EXT_PNG: TYPE_PNG,
    EXT_GIF: TYPE_GIF,
    EXT_ICO: TYPE_GIF,
    EXT_JPG: TYPE_JPG,
    EXT_JPEG: TYPE_JPG,
    EXT_PDF: TYPE_PDF,
    EXT_SVG: TYPE_SVG
  }
  for (var key in types){
    if (_url.endsWith(key)) {
      return types[key];
    }
  }
  return TYPE_PLAIN;
}

var shell = http.createServer(function(request,
response) {
#ifdef DEBUG
  console.log('received request: ', request.url);
#endif
  var filepath = CURRENT_DIR + request.url;
  if (filepath == NO_EXPLICIT_FILE) {
      filepath= filepath + INDEX_HTML;
  }
  fs.readFile(filepath, function(error,
filecontent) {
    if (!error){
      response.writeHead(JS_HTTP_STATUS_OK,
JS_HTTP_HEADER(getType(filepath)));
      response.end(filecontent, JS_CHAR_UTF8);
    }
  });
}).listen(PORT_SOCKET_IO);
```

```
function create_event(socket, event_id){
  socket.emit(event_id);
}

function create_message(socket, event_id, msg){
  socket.emit(event_id, msg);
}

var io = socketio.listen(shell);

io.sockets.on(IO_EVENT_CONNECT, function (socket){
#ifdef DEBUG
  console.log("connected from a browser\n");
#endif

   socket.on(IO_EVENT_COMMAND, function(msg){
#ifdef DEBUG
      console.log("Command Received:"+msg+" \n");
#endif
    out_queue.rpush(REDIS_TO_AI, NAME_KAONASHI +
msg);
  });

  socket.on(IO_EVENT_DISCONNECT, function(){});

});


/* Redis を介した AI エンジンとのインターフェース */

var in_queue1 =
require(JS_REQ_REDIS).createClient();
var in_queue2 =
require(JS_REQ_REDIS).createClient();
var out_queue =
require(JS_REQ_REDIS).createClient();

in_queue1.flushdb();

in_queue1.blpop(REDIS_TO_WEB1, REDIS_NO_TIMEOUT,
show_master);
in_queue2.blpop(REDIS_TO_WEB2, REDIS_NO_TIMEOUT,
show_robot);

function show(event_id, message){
#ifdef DEBUG
  console.log("received", event_id, message);
#endif
  create_message(io.sockets, event_id, message);
}

function show_master(err, msg){
  current_title = msg[REDIS_DATA];
  show(IO_EVENT_MASTER, msg[REDIS_DATA]);
  in_queue1.blpop(REDIS_TO_WEB1, REDIS_NO_TIMEOUT,
show_master);
}

function show_robot(err, msg){
  current_data = msg[REDIS_DATA];
  show(IO_EVENT_ROBOT, msg[REDIS_DATA]);
  in_queue2.blpop(REDIS_TO_WEB2, REDIS_NO_TIMEOUT,
show_robot);
}
```

7.6.3favicon.ico

あまり Web サーバー設計の教科書に書かれていないのですが、http 経由で要求されるファイルのなかに favicon.ico という画像ファイルがあります。これはブラウザのタブに表示するアイコンで、16×16 画素くらいの画像です。index.html と同じディレクトリに置いておきます。

どんな画像でも構わないので、自作してみました（右図）。カオナシの写真を四角く切り出し（背景は透明にしました）、16×16 画素に縮小したら、ico ファイルに書き出します。私は Windows でも Ubuntu でも使える GIMP という画像処理ソフトウエアを使用していますが、これで ico ファイルを作ることができました。
Windows 付属のペイントしかない人は、GIMP などのフリーウエアを探すか、無償で使える ico ファイルの中から適当なものを選んでください。

7.6.4HTML 文書

HTML 文書ファイルも cpp で前処理する前提で設計しました。Index.html.source というファイルを用意し、cpp で処理したものを index.html として使用します。

Socket.IO から受信したメッセージをそれぞれのフィールドに表示します。今回はプロトタイプなので、あまり表示には凝らず、命令とロボットの応答をそれぞれ一行だけ表示することにしました。

入力欄にテキストを書き込み、「コマンド送信」ボタンをクリックすれば、Socket.IO に送り出します。やがて同じテキストが AI エンジンから返されてくるので、上記の要領で表示します。

index.html.source

```
/* ブラウザからの操作用 index.html ファイルのソース
   初版；2020/3/19 Chuji
   最新版：

   注：使用する前に cpp で処理すること
   $ cpp index.html.source | python cleanfile.py
>index.html
*/

#include "./include/web_if.h"

/* html 宣言部 */
<!DOCTYPE html>

<html lang="ja">
<meta charset="utf-8">

<head>
<title>Secretary Robot Kaonashi</title>

<script SCRIPT_SOCKETIO></script>

<script type="text/javascript">

/* Socket.IO 用ソケット生成 */
var mysock =  io.connect(URL_SOCKET_IO);

/* Socket.IO からの受信処理 */
mysock.on(IO_EVENT_CONNECTED, function(){
#ifdef DEBUG
  obj=document.getElementById(WEB_MESSAGE_RAW);
  obj.textContent="connected to the host";
#endif
});

/* 表示要求の処理 */
/* マスターのコマンド */
mysock.on(IO_EVENT_MASTER, function(msg){
#ifdef DEBUG
  obj=document.getElementById(WEB_MESSAGE_RAW);
  obj.textContent="Master's words are received:
"+msg;
#endif
```

```
  obj=document.getElementById(WEB_ID_MASTER);
  obj.textContent=msg;
});

/* ロボットの応答 */
mysock.on(IO_EVENT_ROBOT, function(msg){
#ifdef DEBUG
  obj=document.getElementById(WEB_MESSAGE_RAW);
  obj.textContent="Robot's words are received:
"+msg;
#endif
  obj=document.getElementById(WEB_ID_ROBOT);
  obj.textContent=msg;
});

/* コマンド送信処理 */
function send_command(){

text=document.getElementById(WEB_ID_COMMAND).value;
#ifdef DEBUG
  obj=document.getElementById(WEB_MESSAGE_RAW2);
  obj.textContent="Submitted text: "+text;
#endif
  mysock.emit(IO_EVENT_COMMAND, text);
};

</script>
</head>

<body>

/* HTML 主要部 */
<center>
<font WEB_TITLE_FONT><b>WEB_TITLE</b></font>
</center>

#ifdef DEBUG
<p><div id=WEB_MESSAGE_RAW>サーバーからの受信データ
</div></p>
<p><div id=WEB_MESSAGE_RAW2>ブラウザからの送信データ
</div></p>
#endif

/* 会話表示領域 */
<p align="left">
ロボットの応答:<br>
<span id=WEB_ID_ROBOT></span></p>
<br>

<p align="right">
ロボットへの命令:<br>
<span id=WEB_ID_MASTER></span></p>
<br>

/* コマンド入力領域 */
<p>Text input:<br>
  <input type="text" id=WEB_ID_COMMAND size = "64">
  <input type="button" value="コマンド送信"
onClick="send_command()"></p>

</body>
</html>
```

index.html を作っておきます。

```
$ cpp index.html.source | python cleanfile.py
>index.html
```

7.6.5検証

検証は SSH 窓を 3 つ開いて行います。まず一つ目の窓で popper.py を起動して、REDIS_TO_AI に送られてくるメッセージを表示できるようにしておきます。次に二番目の窓で pusher.py を起動し、REDIS_TO_WEB1 へメッセージを送る準備をします。三番目の窓で Node.js を起動したら、PC のブラウザから http://192.168.15.16:50010 にアクセスしてみます。Web ページが表示されましたか？

ブラウザのタブには、先に作ったカオナシのアイコンが表示されています。

```
1: $ python popper.py AI

   カオナシこんにちは

2: $ python pusher.py Web1
   おはようございます

3: $ cpp web-if.js |node
```

ブラウザの入力欄に「こんにちは」と書いて、「コマンド送信」ボタンをクリックすると、第一の窓にそのテキストが（先頭に「カオナシ」というウェイクアップワードが追加されて）表示されます。

次に第二の窓で「おはようございます」と入力すると、ブラウザの「ロボットへの命令」欄に表示されます。AI エンジンはブラウザからの命令をそのまま返すのですが、ここでは敢えて違う言葉にしました。

以上の確認が済んだら、第二の窓のプロセスを（Ctrl-C キーを押して）停止してから、REDIS_TO_WEB2 にメッセージを送ってみます。

```
2: $ python pusher.py Web2

   さようなら
```

ブラウザ画面の「ロボットの応答」欄にメッセージが表示されたら、検証は完了です。

7.7 AI エンジン

AI エンジンの構成は前に検討しましたが、さらにソフトウエアモジュールに整理しなおします。

- AI エンジン冒頭部 kaonashi.py
- 命令解釈モジュール interpreter.py
- 挨拶モジュール greeting.py
- 時間管理モジュール time_keeper.py
- 音声処理モジュール mouth.py
- 時間取得モジュール（関数）get_time.py
- 感情表現モジュール emotion.py

設計はトップダウンで詳細化していきましたが、具体的なプログラムコードの作成と検証はボトムアップ方式をとって、後ろのモジュールから説明します。

最初にプロトタイプのカオナシが解釈する命令と返事の語彙をインクルードファイルで定義しておきます。後半では、定型応答の音声ファイル名とテキストファイル名を定義しています。

AI_commands.h

```
/* 秘書ロボットへのコマンド
   初版：2020/3/19 Chuji
   最新版：
 */

#ifndef __AI_COMMANDS

/* 日本語コマンド */
#define NAME_KAONASHI  'カオナシ'
#define WORD_HELLO  'こんにちは'
#define WORD_SHUTDOWN  'シャットダウン'
#define WORD_BYE  'さようなら'
#define WORD_BYE2  'さよなら'
#define WORD_TODAY  'きょうは'
#define WORD_WHATDAY  '何日'
#define WORD_WHATTIME  '何時'
#define WORD_WEATHER  '今日の天気は'
#define WORD_MY_SCHEDULE  '予定'
#define WORD_LOOKUP  '調べて'
#define WORD_YEAR  '年'
#define WORD_MONTH  '月'
#define WORD_DAY  '日'
#define WORD_AM  '午前'
#define WORD_PM  '午後'
#define WORD_OCLOCK  '時'
#define WORD_MINUTE  '分'
#define WORD_AFTER  '経ったら'
#define WORD_NOTIFY  '教えて'
#define WORD_TELLME  '知らせて'
#define WORD_MEMORIZE  '私を覚えて'

/* 数字*/
#define WORD_ONE  '一'
#define WORD_TWO  '二'
#define WORD_THREE  '三'
#define WORD_FOUR  '四'
#define WORD_FIVE  '五'
#define WORD_SIX  '六'
#define WORD_SEVEN  '七'
#define WORD_EIGHT  '八'
#define WORD_NINE  '九'
#define WORD_TEN  '十'
#define WORD_HUNDRED  '百'
#define WORD_THOUSAND  '千'

#define CHAR_ZERO  '０'
#define CHAR_ONE  '１'
#define CHAR_TWO  '２'
#define CHAR_THREE  '３'
#define CHAR_FOUR  '４'
#define CHAR_FIVE  '５'
#define CHAR_SIX  '６'
#define CHAR_SEVEN  '７'
#define CHAR_EIGHT  '８'
#define CHAR_NINE  '９'

#define NUM_ZERO  '0'
#define NUM_ONE  '1'
#define NUM_TWO  '2'
#define NUM_THREE  '3'
#define NUM_FOUR  '4'
#define NUM_FIVE  '5'
#define NUM_SIX  '6'
#define NUM_SEVEN  '7'
#define NUM_EIGHT  '8'
#define NUM_NINE  '9'

/* 日本語の返事の語彙 */
#define WORD_TODAYIS  '今日は'
#define WORD_TIMEIS  '今'
#define WORD_DESU  'です'
#define WORD_STARTED  'タイマーを起動しました'
#define WORD_EXPIRED  'タイマーの時間が来ました'
#define WORD_LET_U_KNOW  '後にお知らせします'
#define WORD_TIMER  'タイマー'
#define WORD_NUMBER  '番'
#define WORD_SUNDAY  '日曜日'
#define WORD_MONDAY  '月曜日'
#define WORD_TUESDAY  '火曜日'
#define WORD_WEDNESDAY  '水曜日'
#define WORD_THURSDAY  '木曜日'
#define WORD_FRIDAY  '金曜日'
#define WORD_SATURDAY  '土曜日'
#define WORD_HOURS  '時間'
#define WORD_WEATHERIS  '今日の天気は'
#define WORD_SCHEDULEIS  '今日の予定は'
#define WORD_SIR  'さん'
#define WORD_NOTHING  'ありません'
#define WORD_RESULTIS  '結果は'
#define WORD_TURNON  'をオンにします'
#define WORD_TURNOFF  'をオフにします'
#define WORD_BREATH  '、'
#define WORD_PERIOD  '。'

#define WORD_MORNIN  'おはよう'
#define WORD_MORNING  'おはようございます'
#define WORD_EVENING  'こんばんは'
#define WORD_NIGHT  'おやすみなさい'
#define WORD_IS  'です'
#define WORD_JUSTNAME  'カオナシも、ですも要りません'
#define WORD_NOW  'いま'
#define WORD_SCHEDULE  '予定時間になりました'
#define WORD_SHUTTING  'システムをシャットダウンします'
#define WORD_WAIT  'ちょっと待ってください'
#define WORD_WHO  'どなたですか'
#define WORD_YES  'はい'
#define WORD_ANGRY  '怒っていませんか'
#define WORD_CONCERNED  'なにか心配事でもありますか'
#define WORD_HAPPY  'うれしそうですね'
#define WORD_SAD  'なんだか悲しそうですね'
#define WORD_NAME  '名前を教えてください'



/* 定型応答音声ファイル名 */
#define VOICE_DIRECTORY  '/home/chuji/voice/'
#define VOICE_FILE_EXT  '.wav'
#define VOICE_TEXT  'voice'
#define VOICE_AH  'ah.wav'
#define VOICE_AHAH  'ahah.wav'
#define VOICE_BYE  'bye.wav'
#define VOICE_MORNING  'good_morning.wav'
#define VOICE_EVENING  'good_evening.wav'
#define VOICE_NIGHT  'good_night.wav'
#define VOICE_HELLO  'hello.wav'
#define VOICE_IS  'is.wav'
#define VOICE_JUSTNAME  'just_name.wav'
#define VOICE_NOW  'now.wav'
#define VOICE_SCHEDULE  'scheduled_time.wav'
#define VOICE_SHUTTING  'shutdown.wav'
#define VOICE_STARTED  'timer_started.wav'
#define VOICE_EXPIRED  'timer_expired.wav'
#define VOICE_TODAY  'today.wav'
#define VOICE_WAIT  'wait_a_moment.wav'
#define VOICE_WEATHER  'weather.wav'
#define VOICE_WHO  'who_are_you.wav'
#define VOICE_YES  'yes.wav'
#define VOICE_ANGRY  'you_look_angry.wav'
#define VOICE_CONCERNED  'you_look_concerned.wav'
#define VOICE_HAPPY  'you_look_happy.wav'
#define VOICE_SAD  'you_look_sad.wav'
#define VOICE_NAME  'your_name.wav'

/* 定型応答テキストファイル名 */
#define TEXT_DIRECTORY  '/home/chuji/voice/'
#define TEXT_FILE_EXT  '.txt'
#define TEXT_TEXT  'voice'
#define TEXT_BYE  'bye.txt'
#define TEXT_MORNING  'good_morning.txt'
```

```
#define TEXT_EVENING  'good_evening.txt'
#define TEXT_NIGHT  'good_night.txt'
#define TEXT_IS  'is.txt'
#define TEXT_HELLO  'hello.txt'
#define TEXT_JUSTNAME  'just_name.txt'
#define TEXT_NOW  'now.txt'
#define TEXT_SCHEDULE  'scheduled_time.txt'
#define TEXT_SHUTTING  'shutdown.txt'
#define TEXT_STARTED  'timer_start_'
#define TEXT_EXPIRED  'timer_expired_'
#define TEXT_TODAY  'today.txt'
#define TEXT_WAIT  'wait_a_moment.txt'
#define TEXT_WEATHER  'weather.txt'
#define TEXT_WHO  'who_are_you.txt'
#define TEXT_YES  'yes.txt'
#define TEXT_ANGRY  'you_look_angry.txt'
#define TEXT_CONCERNED  'you_look_concerned.txt'
#define TEXT_HAPPY  'you_look_happy.txt'
#define TEXT_SAD  'you_look_sad.txt'
#define TEXT_NAME  'your_name.txt'

#define __AI_COMMANDS

#endif
```

AI エンジンの単体検証を行うときは、聴覚サブシステムも発話サブシステムも動かさず、pusher.py と popper.py を使って、Redis インターフェース入出力を調べることにします。

7.7.1感情表現モジュール emotion.py

プロトタイプの感情表現は LED ユニットの発光だけで行うことにします。音声合成で使う声色ファイルで表現することも考えましたが、あまり音色ファイルの選択肢が多くないし、あらかじめ合成しておいた音声を使う必要があるので、諦めました。

設計

感情のバラエティを増やすため、それぞれの感情ごとに操作を設定します。思いついた順番に並べたので、取り留めないと感じるかもしれません。

クラス	emotion	感情表現
属性	strip	LED ユニットドライバ
操作	show	指定色パターンで発光させる
	happy	楽しい気分を表現する
	angry	怒った気分を表現する
	fine	元気な気分を表現する
	serious	真面目な気分を表現する

emotion.py のオブジェクト

インクルードファイル emotion.h には、LED ユニットの構成と、感情を表現する発色パターンを定義しています。いまはデバグのため、感情ごとに異なる LED を発光させているだけです。

emotion.h

```
/* emotion.h  カラーLED による感情表現
   初版： 2020/4/2/ Chuji
   最新版：
*/

/* LED モジュールの構成： */
#define LED_COUNT  8  /* LED 素子数 */
#define LED_PIN    10  /* GPIO ピン番号：SPI MISO */
#define LED_FREQ_HZ  800000  /* LED 通信用周波数：8KHz */
#define LED_DMA    10  /* 使用する DMA チャンネル（推奨） */
#define LED_BRIGHTNESS  12  /* LED 輝度（明るすぎるので落として置く） */
#define LED_INVERT  False  /* 信号極性反転：不要 */
#define LED_CHANNEL  0  /* ハードウエアチャンネル（推奨） */

/* 感情を表現する色（R, B, G）の配列（LED は左から順番）各 0～255 */
#define BLACK    [0, 0, 0]
#define RED    [255, 0, 0]
#define GREEN    [0, 255, 0]
#define BLUE    [0, 0, 255]
#define WHITE    [255, 255, 255]

#define COLOR_HAPPY  [BLACK, BLACK, GREEN, BLACK, BLACK, GREEN, BLACK, BLACK]
#define COLOR_FINE  [BLUE, BLACK, BLACK, BLACK, BLACK, BLACK, BLACK, BLUE]
#define COLOR_ANGRY  [BLACK, BLACK, RED, RED, RED, RED, BLACK, BLACK]
#define COLOR_SERIOUS  [BLACK, BLACK, BLACK, WHITE, WHITE, BLACK, BLACK, BLACK]
```

感情表現モジュール emotion.py は、ほとんど同じ操作の集合体です。初期化と発色を実行する操作を除き、すべて同じ構造（発色パターンを指定して実行部を呼ぶ）をとっています。ひとつの操作で十分なように見えますが、後で複雑な感情を表現するために、発光パターンを時間的に変えるといった手法をとれるようにするために、こうしました。

マクロ定義 BCM2835 を#define してなければ、LED を発光させる代わりにテキストを表示します。Raspberry Pi 以外の環境でも実行できるようにするためです。

emotion.py

```
/* emotion.py  カラーLED による感情表現
   初版： 2020/4/2/ Chuji
   最新版：

Class: emotion
属性:
  strip:  LED ドライバ
操作:
  show:    表情を表現する
  happy:  楽しい気分
  angry:  怒った気分
  fine:    元気な気分
  serious:  真面目な気分
*/

#include "include/emotion.h"

from rpi_ws281x import PixelStrip, Color

/* 感情表示オブジェクト */

class emotion:
  def __init__(self):
#ifdef BCM2835
```

```
    /* LED ドライバーソフトの開始 */
    self.strip = PixelStrip(LED_COUNT, LED_PIN,
LED_FREQ_HZ, LED_DMA, LED_INVERT, LED_BRIGHTNESS,
LED_CHANNEL)
    self.strip.begin()
#else
    print('Emotion module is initiated.')
#endif

#ifdef BCM2835
  def show(self, c):
    for i in range(LED_COUNT):
      self.strip.setPixelColor(i, Color(c[i][0],
c[i][1], c[i][2]))
    self.strip.show()
#endif

  def happy(self):
#ifdef BCM2835
    self.show(COLOR_HAPPY)
#else
    print('EMOTION: I am happy.')
#endif

  def angry(self):
#ifdef BCM2835
    self.show(COLOR_ANGRY)
#else
    print('EMOTION: I am angry.')
#endif

  def fine(self):
#ifdef BCM2835
    self.show(COLOR_FINE)
#else
    print('EMOTION: I am fine.')
#endif

  def serious(self):
#ifdef BCM2835
    self.show(COLOR_SERIOUS)
#else
    print('EMOTION: I am serious.')
#endif
```

検証

検証は、すべての感情表現操作を呼び出すスタブを使って行います。

test_emotion.py

```
/* test_emotion.py  感情表現モジュールの検証
   初版： 2020/4/2/ Chuji
   最新版：
*/

#include "include/use-time.h"

#include "emotion.py"

/* 感情表示オブジェクト */
face = emotion()

#define WAITING  5

print('元気な気分')
face.fine()
sleep(WAITING)

print('楽しい気分')
face.happy()
sleep(WAITING)

print('真面目な気分')
face.serious()
sleep(WAITING)

print('怒った気分')
face.angry()
```

最初に BCM2835 を#define せずに検証すれば、それぞれの感情を表すテキストが表示されます。
Raspberry Pi 上で、BCM2835 を#define して実行すれば、LED が指定パターンどおりに発光します。ここでは、思い通りのパターンで発光させられることだけを確認します。発光パターンが感情を良く表現しているかどうかは、検証の対象外です。

7.7.2時間取得モジュール get_time.py

AI の命令解釈の一環として、日本語テキストから時間表現を取り出す関数を用意しました。「時間」あるいは「分」を探し、その前にある数字を数値に変換して、分単位で返します。

設計

時刻を表す数値の形式には、以下の三種類があります。Julius が返してくる時間表現は漢数字ですが、ブラウザから命令を与えることも考慮し、すべての形式を処理できるようにします。

- 半角数字（0～9）
- 全角数字（０～９）
- 漢数字（〇～九、十）

半角数字は ASCII コードなので、文字は数値順に並んでいます。C 言語の教科書には、文字コードを引き算する（"5" - "0"）ことで数値 5 を求める方法が書かれています。全角数字も同じ順番で並んでいますが、漢数字は（部首と画数で整列しているため）同じ手法が使えません。そこで、辞書 ch2num で数値に直す工夫をしました。

漢数字を先頭から解釈していくときには「二十」と「十」で処理を変えなければいけません。「十」の後に漢数字が出てくる場合も特別な処理がいります。「百」も含む場合を考え出したのですが、あまりに面倒だったので止めました。

その代わり「二時間半」は処理できるようにしました。「一分半」の「半」は無視されます。

get_time.py

```
/* テキストから時間を取り出し、分単位で返す（見つからなければゼロを返す）
      初版： 2020/3/24 Chuji
      最新版：

検出範囲：半角数字、全角数字、漢数字　＋　「時」、「分」、「半」
          数値部は百未満を仮定する。
          秒は含まれないと仮定する。
*/
```

```
nums = ['0', '1', '2', '3', '4', '5', '6', '7',
'8', '9', '０', '１', '２', '３', '４', '５', '６', '
７', '８', '９', '〇', '一', '二', '三', '四', '五', '
六', '七', '八', '九', '十', '半', '時', '分']

ch2num = {'0':0, '1':1, '2':2, '3':3, '4':4, '5':5,
'6':6, '7':7, '8':8, '9':9, '０':0, '１':1, '２':2,
'３':3, '４':4, '５':5, '６':6, '７':7, '８':8, '９
':9, '〇':0, '一':1, '二':2, '三':3, '四':4, '五':5,
'六':6, '七':7, '八':8, '九':9}

def get_time(text):
  dat = 0    /* 解析中の数値 */
  dat2 = 0  /* 時間の数値（分単位） */
  after10 = False  /* 漢数字'十'の直後の特別処理フラグ
*/

  for pos in range(len(text)):
    if text[pos] not in nums:
      pass
    elif text[pos] in ch2num:
      if after10:
        dat = dat + ch2num[text[pos]]
      else:
        dat = 10 * dat + ch2num[text[pos]]
      after10 = False
    elif text[pos] == '十':  /* 百以上の桁はないとする
*/
      if dat != 0:  /* 二十～九十 */
        dat = 10 * dat
      else:    /* 十 */
        dat = 10
      after10 = True
    elif text[pos] == '時':  /* 「時」と「時間」を兼用
*/
      dat2 = 60 * dat
      dat = 0
    elif text[pos] == '分':  /* 検出終了 */
      return dat2 + dat
    elif text[pos] == '半':
      if (dat2 != 0) and (dat == 0):  /* ○○時間半
*/
        return dat2 + 30
    else:
      pass
  if (dat2 != 0) and (dat == 0):  /* ○○時間 */
    return dat2
  else:        /* ○○時間○○  または  ○○ */
    return 0
```

検証

検証プログラム test_gettime.py を用意し、日本語テキストを入力して、変換結果を表示します。

test_gettime.py

```
/* 関数 get_time の評価プログラム
   初版：2020/3/29 Chuji
  最新版：
*/

#include "get_time.py"

try:
  while True:
    text = input()

    print('Text:', text, 'is translated to:',
get_time(text), 'min.')

except KeyboardInterrupt:
  pass
```

この検証プログラムは標準入力を使うので、前にやったように、cpp の結果をいったん test.py に収納してから、このファイルを実行します

```
$ cpp test_gettime.py | python cleanfile.py
>test.py; python test.py
一時間
Text: 一時間 is translated to: 60 min.
:
```

キーボードから日本語テキストを入力すると、分を単位とした時間に変換していることが確認できます。検証のカバレッジ（検証する範囲）を確保するため、日本語テキストファイルを作成しておき、これを標準入力に与えます。評価が自動で行われるので、何度でも繰り返して確認できます。日本語テキストには、数値の想定範囲内外、例外処理などを網羅するようにします。漢数字と全半角数字の混在は仕様の範囲に含んでいません。

test_gettime.txt（一部のみ掲載）

```
これはダミー
0 分
1 分
:
一時間
:
```

```
$ cpp test_gettime.py | python cleanfile.py
>test.py; python test.py <test_gettime.txt
```

7.7.3音声処理モジュール mouth.py

発話サブシステムとのインターフェースを担うモジュールです。命令を Web にエコーバックするのと、返答を Web に表示する機能も付けてあります。

設計

多くの操作が Web への表示を伴うので、入力パラメータに表示テキストを含ませました。定型の返事をする場合、テキストから音声ファイル名を検索する辞書 voices を保持させています。

実際の処理は、Redis にデータを送るだけなので、Redis キーなどを変えたマクロ定義を多用することにし、ファイル冒頭にまとめています。

クラス	speech_interface	発話サブシステムとのインターフェース
属性	fifo	Redis へのアクセス
	voices	定型の返事と音声ファイルの対照表
操作	ah	返事の最初の言葉を言う
	ahah *)	返事の最後の言葉を言う

	say	定型の返事をキューに入れる
	say_now *)	定型の返事をすぐに言う
	say_this *)	与えられた返事をする
	speak_now *)	キューにある音声をすべて再生する
	synthesize	音声を合成して返事をする
	repeat	命令を復唱する（Web 表示のみ）

mouth.py のオブジェクト

操作のうち、*）印が付いているものは、最後に音声再生クライアントに play を指示します。

mouth.py

```
/*  (AI エンジン）発語サブシステム/WEB とのインターフェース
      初版：  2020/3/21 Chuji
      最新版：

クラス：
  speech_interface:
属性：
  fifo   : Redis FIFO
  voice   : 定型テキストと音声ファイルの対照表
操作：
  ah     :返事の最初にアーッという
  ahah     :返事の最後にアッアーという（発語）
  say     :定型の返事をキューに入れる
  say_now   :定型の返事をする（即時発語）
  say_this   :与えられた返事する（テキストと音声ファイルを与
える）
  speak_now   :キューにある返事をまとめて発語する
  synthesize   :返事を音声合成する
  repeat   :命令を復唱する（Web 表示のみ）

*/

#include "include/use-redis.h"
#include "include/mpc.h"
#include "include/AI_commands.h"

/* REDIS FIFO へ指示を送るマクロ定義（このファイルでのみ使
用)*/
#define ADD_FILE(x)  rpush(REDIS_TO_MPC, MPC_ADD +
x)
#define PLAY_FILE(x)  rpush(REDIS_TO_MPC,
MPC_IMMEDIATE + x)
#define MASTER_SAYS(x)  rpush(REDIS_TO_WEB1, x)
#define ROBOT_SAYS(x)  rpush(REDIS_TO_WEB2, x)
#define WAIT_SYNTHESIS  rpush(REDIS_TO_MPC,
MPC_WAIT)
#define PLAY_ALL  rpush(REDIS_TO_MPC, MPC_PLAY)
#define SYNTHESIS(x)  rpush(REDIS_TO_OPENJ, x)


class speech_interface:
  def __init__(self, redis_fifo):
    self.fifo = redis_fifo
    self.voices =
{WORD_BYE:VOICE_BYE,WORD_MORNING:VOICE_MORNING,
WORD_EVENING:VOICE_EVENING, WORD_NIGHT:VOICE_NIGHT,
WORD_HELLO:VOICE_HELLO, WORD_IS:VOICE_IS,
WORD_JUSTNAME:VOICE_JUSTNAME, WORD_NOW:VOICE_NOW,
WORD_SCHEDULE:VOICE_SCHEDULE,
WORD_SHUTTING:VOICE_SHUTTING,
WORD_STARTED:VOICE_STARTED,
WORD_EXPIRED:VOICE_EXPIRED, WORD_TODAY:VOICE_TODAY,
WORD_WAIT:VOICE_WAIT, WORD_WEATHER:VOICE_WEATHER,
WORD_WHO:VOICE_WHO, WORD_YES:VOICE_YES,
WORD_ANGRY:VOICE_ANGRY,
WORD_CONCERNED:VOICE_CONCERNED,
WORD_HAPPY:VOICE_HAPPY, WORD_SAD:VOICE_SAD,
WORD_NAME:VOICE_NAME}


  def ah(self):  /* 返事の最初 */
    self.fifo.ADD_FILE(VOICE_AH)  /* 次の返事を待って
から発語する */

  def ahah (self):  /* 返事の最後 */
    self.fifo.PLAY_FILE(VOICE_AHAH)  /* すべて発語し
て、次の命令を待つ */

  def say (self, text):  /* 定型の返事をキューに入れる
*/
    if(text in self.voices):
      voice = self.voices[text]
      self.fifo.ROBOT_SAYS(text)  /* 返事を Web に表示
*/
      self.fifo.ADD_FILE(voice)  /* 返事を発語キューに
入れる */

  def say_now (self, text):  /* 定型の返事をすぐに発語
する */
    if (text in self.voices):
      voice = self.voices[text]
      self.fifo.ROBOT_SAYS(text)  /* 返事を Web に表示
する */
      self.fifo.PLAY_FILE(voice)  /* 返事を発語する
*/

  def speak_now (self):  /* キューにある返事をまとめて発
語する */
    self.fifo.PLAY_ALL

  def say_this (self, text, voice):  /* 与えられた返
事をする */
    self.fifo.ROBOT_SAYS(text)  /* テキストを Web に表
示する */
    self.fifo.ADD_FILE(voice)  /* 音声ファイルをすぐ再
生する */


  def synthesize (self, text):  /* 返事を音声合成する
*/
    self.fifo.ROBOT_SAYS(text)  /* テキストを Web に表
示する */
    self.fifo.WAIT_SYNTHESIS  /* 音声合成待ちの場所を確
保する */
    self.fifo.SYNTHESIS(text)  /* 音声合成を行う */

  def repeat(self, text):  /* 命令を復唱する（Web 表示
のみ） */
    self.fifo.MASTER_SAYS(text)  /* 命令を Web に表示す
る */
```

検証

mouth.py は単純な機能の操作ばかりなので、それぞれを実行して検証します。また辞書の全項目と、辞書にない項目の処理（何もしない）を確認もします。そのためのプログラム test_mouth.py を用意しました。

test_mouth.py

```
#include "include/use-time.h"
#include "mouth.py"

fifo = open_FIFO()
mouth = speech_interface(fifo)
print ('カオナシの発声')
input('push return to go')
mouth.ah()
mouth.ahah()

print('すぐに発話する -say_now')
input('push return to go')
mouth.say_now('おはようございます')

print('キュー入れる-say')
input('push return to go')
```

```
mouth.say('こんにちは')
mouth.say('さようなら')

print('キューを再生する-speak_now')
input('push return to go')
mouth.speak_now()

print('任意のテキストとファイルの組み合わせ -say_this')
input('push return to go')
mouth.say_this('どうでもいいや', 'now.wav')

print('音声合成指令 - synthesize')
input('push return to go')
mouth.synthesize('何でも言えますよ')
mouth.speak_now()

print('コマンドの Web 表示 -repeat')
input('push return to go')
mouth.repeat('与えられた命令')

#define WAITING  3
print('辞書の範囲検証 - say_now')
input('push return to go')
mouth.say_now(WORD_BYE)
sleep(WAITING)
mouth.say_now(WORD_MORNING)
sleep(WAITING)
mouth.say_now(WORD_NIGHT)
sleep(WAITING)
mouth.say_now(WORD_HELLO)
sleep(WAITING)
mouth.say_now(WORD_IS)
sleep(WAITING)
mouth.say_now(WORD_JUSTNAME)
sleep(WAITING)
mouth.say_now(WORD_SHUTTING)
sleep(WAITING)
mouth.say_now(WORD_STARTED)
sleep(WAITING)
mouth.say_now(WORD_EXPIRED)
sleep(WAITING)
mouth.say_now(WORD_TODAY)
sleep(WAITING)
mouth.say_now(WORD_WAIT)
sleep(WAITING)
mouth.say_now(WORD_WEATHER)
sleep(WAITING)
mouth.say_now(WORD_WHO)
sleep(WAITING)
mouth.say_now(WORD_YES)
sleep(WAITING)
mouth.say_now(WORD_ANGRY)
sleep(WAITING)
mouth.say_now(WORD_CONCERNED)
sleep(WAITING)
mouth.say_now(WORD_HAPPY)
sleep(WAITING)
mouth.say_now(WORD_SAD)
sleep(WAITING)
mouth.say_now(WORD_NAME)

print('辞書に存在しない言葉')
input('press return to go')
mouth.say_now('ありえない言葉')
```

mouth.py の出力は全て Redis に対して行われるので、REDIS_TO_MPC、REDIS_TO_OPENJ、REDIS_TO_WEB1、REDIS_TO_WEB2 の 4 つのキーをすべて popper.py で監視します。

1:
```
$ cpp test_mouth.py | python cleanfile.py
>test.py
$ python test.py
カオナシの発声
push return to go
すぐに発話する -say_now
push return to go
キュー入れる-say
push return to go
キューを再生する-speak_now
push return to go
任意のテキストとファイルの組み合わせ -say_this
push return to go
音声合成指令 - synthesize
push return to go
コマンドの Web 表示 -repeat
push return to go
辞書の範囲検証 - say_now
push return to go
辞書に存在しない言葉
press return to go
$
```

2:
```
$ python popper.py mpc & python popper.py
Web1 & python popper.py Web2 & python
popper.py OpenJTalk

Text: Aah.wav from: mpc
Text: Iahah.wav from: mpc
Text: おはようございます from: Web2
Text: Igood_morning.wav from: mpc
Text: こんにちは from: Web2
Text: さようなら from: Web2
Text: Ahello.wav from: mpc
Text: Abye.wav from: mpc
Text: P from: mpc
Text: どうでもいいや from: Web2
Text: Inow.wav from: mpc
Text: 何でも言えますよ from: Web2
Text: 何でも言えますよ from: OpenJTalk
Text: W from: mpc
Text: P from: mpc
Text: 与えられた命令 from: Web1
Text: さようなら from: Web2
Text: Ibye.wav from: mpc
Text: おはようございます from: Web2
Text: Igood_morning.wav from: mpc
Text: おやすみなさい from: Web2
Text: Igood_night.wav from: mpc
Text: こんにちは from: Web2
Text: Ihello.wav from: mpc
Text: です from: Web2
Text: Iis.wav from: mpc
Text: カオナシも、ですも要りません from: Web2
Text: Ijust_name.wav from: mpc
Text: システムをシャットダウンします from: Web2
Text: Ishutdown.wav from: mpc
Text: タイマーを起動しました from: Web2
Text: Itimer_start_ from: mpc
Text: タイマーの時間が来ました from: Web2
Text: Itimer_expired_ from: mpc
Text: 今日 from: Web2
Text: Itoday.wav from: mpc
Text: ちょっと待ってください from: Web2
Text: Iwait_moment.wav from: mpc
Text: 天気 from: Web2
Text: Iweather.wav from: mpc
Text: どなたですか from: Web2
Text: Iwho_are_you.wav from: mpc
Text: はい from: Web2
Text: Iyes.wav from: mpc
Text: 怒っていませんか from: Web2
Text: Iyou_look_angry.wav from: mpc
Text: なにか心配事でもありますか from: Web2
Text: Iyou_look_concerned.wav from: mpc
Text: うれしそうですね from: Web2
Text: Iyou_look_happy.wav from: mpc
Text: なんだか悲しそうですね from: Web2
Text: Iyou_look_sad.wav from: mpc
Text: 名前を教えてください from: Web2
Text: Iyour_name.wav from: mpc
```

Redis への出力が予想どおりだったので、こんどは REDIS_TO_WEB1 と REDIS_TO_WEB2 だけを監視しながら、発話サブシステムを起動して試してみます。

1:
```
$ cpp test_mouth.py | python cleanfile.py
>test.py
$ python test.py
（後略）
```

2:
```
$  python popper.py Web1 & python popper.py
Web2 & ./run_voice
再生中 WAVE '/home/chuji/voice/ah.wav' :
Signed 16 bit Little Endian, レート 48000 Hz,
ステレオ
再生中 WAVE '/home/chuji/voice/ahah.wav' :
Signed 16 bit Little Endian, レート 48000 Hz,
ステレオ
Text: おはようございます from: Web2
再生中 WAVE
'/home/chuji/voice/good_morning.wav' : Signed
16 bit Little Endian, レート 48000 Hz, モノラル
Text: こんにちは from: Web2
Text: さようなら from: Web2
再生中 WAVE '/home/chuji/voice/hello.wav' :
Signed 16 bit Little Endian, レート 48000 Hz,
モノラル
再生中 WAVE '/home/chuji/voice/bye.wav' :
Signed 16 bit Little Endian, レート 48
:
（後略）
```

REDIS_TO_WEB2 へ表示された音声が（任意のテキスト音声の組み合わせのケースを除き）再生されることを確認すれば、音声処理モジュールの検証は終わりです。

ここでいったん発話サブシステムを停止させ、この章の残りのモジュールの検証は Redis インターフェースで行います。

7.7.4時間管理 モジュール time_keeper.py

時間管理モジュールでは、今日の日付や現在時刻を言ったり、現在時刻を返したり、タイマーを起動したりします。

設計

日時を求めるための定義をインクルードファイル use-datetime.h にまとめました。

use_datetime.h

```
/* use-datetime.h  日付・時刻取得
  初版：2020/4/2 Chuji
  最新版：
*/

#ifndef __USE_TIMER

from datetime import datetime
import locale

#define JAPAN  setlocale(locale.LC_TIME,
'ja_JP.UTF-8')
#define GET_YEAR(x)  x.strftime('%Y')+WORD_YEAR
#define GET_MONTH(x)  x.strftime('%B')
#define GET_DAY(x)  x.strftime('%d')+WORD_DAY
#define GET_WEEK(x)  x.strftime('%A')
#define GET_24HOUR(x)  x.strftime('%H')
#define GET_HOUR(x)  x.strftime('%I')+WORD_OCLOCK
#define GET_MINUTE(x)  x.strftime('%M')+WORD_MINUTE
#define GET_AMPM(x)  x.strftime('%p')
#define IS_ZERO(x)  x[0]=='0'
#define TRIM(x)    x[1:]

#define __USE_TIMER

#endif
```

このほか、タイマーサブシステムで定義した timer.h も使います。

オブジェクト定義は下表のとおりです。

クラス	time_keeper	時間管理機能
属性	mouth	発話インターフェースオブジェクト
	timer	使用するタイマー番号
	voices	タイマー起動報告音声のリスト
操作	tell_date	今日の日付を答える
	tell_time	現在時刻を答える
	get_hour	現在時刻（時間）を得る
	run_timer	タイマーを起動する

time_keeper.py のオブジェクト

日付と時刻は、システムから取得してから、テキストに再構成します。Web に表示するとともに、音声合成します。

get_hour は現在時刻を知って、挨拶の種類を変えるために使います。

run_timer では、起動したタイマー番号を言い、タイマープロセスを起動したあと「〇分後にお知らせします」と言うようにしました。タイマー番号は 1 ずつ増やしていき、上限に達したら初期化します。

time_keeper.py

```
/* 時刻・時間に関する業務実行
     初版： 2020/3/20 Chuji
     最新版：

クラス：
  time_keeper
属性：
  mouth: 発話インターフェースオブジェクト
  timer: 次に使用するタイマーの番号
  voices:タイマー起動報告音声ファイルのリスト
操作：
  tell_date:  今日の日付を言わせる
  tell_time:  現在時刻を言わせる
  get_hour:   現在時刻（時間）を得る
  run_timer:  タイマーを起動する
*/

#include "include/use-datetime.h"
#include "include/timer.h"
#include "include/use-sys.h"
#include "include/AI_commands.h"

class time_keeper:
  def __init__(self, mouth):
    self.mouth = mouth
    locale.JAPAN
    self.timer = 1
    self.voices = TIMER_STARTED

  /* 日付を答える */
#ifdef SIMULATION
  def tell_date(self, ctime):
```

```
#else
  def tell_date(self):
    ctime = datetime.now()
#endif
    year = GET_YEAR(ctime)
    month = GET_MONTH(ctime)
    day = GET_DAY(ctime)
    if(IS_ZERO(day)):
      day = TRIM(day)
    week = GET_WEEK(ctime)
    reply = WORD_TODAYIS + year + month + day +
WORD_BREATH + week + WORD_IS
#ifdef DEBUG
    print(reply)
#else
    self.mouth.synthesize(reply)
#endif

  /* 現在時刻を答える */
#ifdef SIMULATION
  def tell_time(self, ctime):
#else
  def tell_time(self):
    ctime = datetime.now()
#endif
    ampm = GET_AMPM(ctime)
    hour = GET_HOUR(ctime)
    if(IS_ZERO(hour)):
      hour = TRIM(hour)
    minute = GET_MINUTE(ctime)
    if(IS_ZERO(minute)):
      minute = TRIM(minute)

    reply = WORD_TIMEIS + WORD_BREATH + ampm + hour
+ minute + WORD_IS
#ifdef DEBUG
    print(reply)
#else
    self.mouth.synthesize(reply)
#endif

  /* 現在時刻を得る */
#ifdef SIMULATION
  def get_hour(self, ctime):
#else
  def get_hour(self):
    ctime = datetime.now()
#endif
    hour = int(GET_24HOUR(ctime))
    return(hour)


  /* タイマーを起動する */
  def run_timer (self, minute):
    self.mouth.ah()

self.mouth.say_this(str(self.timer)+TIMER_START_MES
SAGE, self.voices[self.timer - 1])

    time_sec = str(60 * minute)
    process = START_TIMER(str(self.timer),
time_sec)
#ifdef DEBUG
    print(process)
#else
    fork(process)
#endif
    message = str(minute) + WORD_MINUTE +
WORD_LET_U_KNOW +WORD_PERIOD
    self.mouth.synthesize(message)
    self.mouth.ahah()

    self.timer += 1
    if self.timer > AVAIL_TIMERS:
      self.timer = 1
```

検証

検証は次の 3 ステップで行います。

1. システム時計から今日の日付と現在時刻を知る
2. SIMULATION モードを使って、与える時間を変え、書式が目的どおりか検証する
3. タイマーを起動させる

まず、test_time_keeper1.py でシステム時計を使ってみます。

test_time_keeper1.py

```
/*  test_time_keeper1.py 現在のシステム時計を使う
  初版：2020/4/2 Chuji
  最終版：
*/

#include "mouth.py"
#include "time_keeper.py"

fifo = open_FIFO()
mouth = speech_interface(fifo)
watch = time_keeper(mouth)

watch.tell_date()

watch.tell_time()

print(watch.get_hour())
```

第一の SSH 窓で test_time_keeper1.py を起動し、第二 SSH 窓で Redis の REDIS_TO_MPC と REDIS_TO_WEB2 を監視します。Web には今日の日付と現在時刻が表示され、音声再生クライアントには WAIT と PLAY 命令が送られてきています。音声合成エンジンは動いていないので、合成結果は送られてきていません。第一の SSH 窓には get_time で得られた現在時刻（24 時間表示の数値）が表示されます。

```
1:  $ cpp test_time_keeper1.py |python
    17

2:  $ python popper.py mpc & python popper.py
    Web2
    Text: 今日は 2020 年 4 月 3 日、金曜日です from:
    Web2
    Text: W from: mpc
    Text: 今、午後 5 時 41 分です from: Web2
    Text: W from: mpc
```

つぎに、SIMULATION モードにして、システム時計出力をいろいろ変えてみます。これは、時刻が’05’などのテキストで返された場合、ゼロを消す処理が正常に動いているか確認することを目的としています。

test_time_keeper2.py

```
/*  test_time_keeper2.py  datetime の値を変えてゼロサプ
レスを確認する
  初版：2020/4/2 Chuji
  最終版：
*/

#define SIMULATION
/*
#define DEBUG
*/
#include "mouth.py"
```

```
#include "time_keeper.py"

fifo = open_FIFO()
mouth = speech_interface(fifo)
watch = time_keeper(mouth)

ctime = datetime(2020, 4, 2, 5, 20, 0, 0)
print(ctime)

watch.tell_date(ctime)

watch.tell_time(ctime)

print(watch.get_hour(ctime))

ctime = datetime(2011, 3, 11, 10, 5, 0)
print(ctime)

watch.tell_date(ctime)

watch.tell_time(ctime)

print(watch.get_hour(ctime))

ctime = datetime(1964, 11, 29, 21, 55, 0, 0)
print(ctime)

watch.tell_date(ctime)

watch.tell_time(ctime)

print(watch.get_hour(ctime))

ctime = datetime(2060, 1, 31, 0, 0, 0, 0)
print(ctime)

watch.tell_date(ctime)

watch.tell_time(ctime)

print(watch.get_hour(ctime))
```

第二の SSH 窓で popper.py が監視を続けていることを前提としています。もし停止していたら、もう一度立ち上げてください。

```
1:  $ cpp test_time_keeper2.py |python
    2020-04-02 05:20:00
    5
    2011-03-11 10:05:00
    10
    1964-11-29 21:55:00
    21
    2060-01-31 00:00:00
    0

2:  Text: 今日は 2020 年 4 月 2 日、木曜日です from:
    Web2
    Text: W from: mpc
    Text: 今、午前 5 時 20 分です from: Web2
    Text: W from: mpc
    Text: 今日は 2011 年 3 月 11 日、金曜日です from:
    Web2
    Text: W from: mpc
    Text: 今、午前 10 時 5 分です from: Web2
    Text: W from: mpc
    Text: 今日は 1964 年 11 月 29 日、日曜日です from:
    Web2
    Text: W from: mpc
    Text: 今、午後 9 時 55 分です from: Web2
    Text: W from: mpc
    Text: 今日は 2060 年 1 月 31 日、土曜日です from:
    Web2
    Text: W from: mpc
    Text: 今、午前 12 時 0 分です from: Web2
    Text: W from: mpc
```

日、時、分の数値が一桁のとき、ゼロが消えていることが確認できました。

最後にタイマー起動を検証します。適当に間隔をとりながら、複数のタイマーを起動してみましょう。

test_time_keeper3.py

```
/*  test_time_keeper3.py タイマーを起動する
  初版：2020/4/3 Chuji
  最終版：
*/

#include "include/use-time.h"
#include "mouth.py"
#include "time_keeper.py"

fifo = open_FIFO()
mouth = speech_interface(fifo)
watch = time_keeper(mouth)

watch.run_timer(1)
sleep(2)
watch.run_timer(1)
sleep(2)
watch.run_timer(2)
sleep(60)
watch.run_timer(1)
sleep(2)
watch.run_timer(1)
sleep(2)
watch.run_timer(1)
```

いったん killall -9 python ですべての監視プログラムを停止してから、改めて 3 つの Redis FIFO を監視します。実行結果は次のとおりです。

```
1:  $ cpp test_time_keeper3.py | python

2:  $ python popper.py mpc & python popper.py
    Web2 & python popper.py OpenJTalk
    Text: Aah.wav from: mpc
    Text: Aah.wav from: mpc
    Text: 1 番タイマーを起動しました。 from: Web2
    Text: Atimer_start_1.wav from: mpc
    Text: 1 分後にお知らせします。 from: Web2
    Text: W from: mpc
    Text: 1 分後にお知らせします。 from: OpenJTalk
    Text: Iahah.wav from: mpc
    Text: Aah.wav from: mpc
    Text: 2 番タイマーを起動しました。 from: Web2
    Text: Atimer_start_2.wav from: mpc
    Text: 1 分後にお知らせします。 from: Web2
    Text: W from: mpc
    Text: 1 分後にお知らせします。 from: OpenJTalk
    Text: Iahah.wav from: mpc
    Text: Aah.wav from: mpc
    Text: 3 番タイマーを起動しました。 from: Web2
    Text: Atimer_start_3.wav from: mpc
    Text: 2 分後にお知らせします。 from: Web2
    Text: W from: mpc
    Text: 2 分後にお知らせします。 from: OpenJTalk
    Text: Iahah.wav from: mpc
    Text: 1 番タイマーの時間が来ました。 from: Web2
    Text: Aah.wav from: mpc
    Text: Atimer_expired_1.wav from: mpc
    Text: Iahah.wav from: mpc
    Text: 2 番タイマーの時間が来ました。 from: Web2
    Text: Aah.wav from: mpc
    Text: Atimer_expired_2.wav from: mpc
    Text: Iahah.wav from: mpc
    Text: Aah.wav from: mpc
    Text: 4 番タイマーを起動しました。 from: Web2
    Text: Atimer_start_4.wav from: mpc
    Text: 1 分後にお知らせします。 from: Web2
    Text: W from: mpc
    Text: 1 分後にお知らせします。 from: OpenJTalk
    Text: Iahah.wav from: mpc
    Text: Aah.wav from: mpc
    Text: 5 番タイマーを起動しました。 from: Web2
    Text: Atimer_start_5.wav from: mpc
    Text: 1 分後にお知らせします。 from: Web2
    Text: W from: mpc
```

```
Text: 1 分後にお知らせします。 from: OpenJTalk
Text: Iahah.wav from: mpc
Text: Aah.wav from: mpc
Text: 1 番タイマーを起動しました。 from: Web2
Text: Atimer_start_1.wav from: mpc
Text: 1 分後にお知らせします。 from: Web2
Text: W from: mpc
Text: 1 分後にお知らせします。 from: OpenJTalk
Text: Iahah.wav from: mpc
Text: 3 番タイマーの時間が来ました。 from: Web2
Text: Aah.wav from: mpc
Text: Atimer_expired_3.wav from: mpc
Text: Iahah.wav from: mpc
Text: 4 番タイマーの時間が来ました。 from: Web2
Text: Aah.wav from: mpc
Text: Atimer_expired_4.wav from: mpc
Text: Iahah.wav from: mpc
Text: 5 番タイマーの時間が来ました。 from: Web2
Text: Aah.wav from: mpc
Text: Atimer_expired_5.wav from: mpc
Text: Iahah.wav from: mpc
Text: 1 番タイマーの時間が来ました。 from: Web2
Text: Aah.wav from: mpc
Text: Atimer_expired_1.wav from: mpc
Text: Iahah.wav from: mpc
```

タイマーを起動したときと、時間が来た時の音声を、Web 表示と音声再生クライアントへの命令で確認します。発話は ah.wav と ahah.wav で挟まれているとと、タイマー番号が使いまわされていることが検証できました。

7.7.5挨拶モジュール greeting.py

挨拶モジュールでは、単純な定型文を言わせます。

クラス	greetings	挨拶機能
属性	mouth	発話インターフェースオブジェクト
操作	start_job	始業の挨拶を言う
	yes	「はい」と返事をする
	end_job	終業の挨拶を言う

time_keeper.py のオブジェクト

始業時には、現在時刻に合わせて挨拶（おはようございます、こんにちは、こんばんは）を使い分けることにしました。

greeting.py

```
/* ロボットの基本的な挨拶
     初版： 2020/3/21 Chuji
     最新版：

クラス：
  greetings
属性：
  mouth: 発語インターフェースオブジェクト
操作：
  start_job: 業務開始の挨拶
  yes:       単純な返事
  end_job:   業務終了の挨拶

*/

#include "include/AI_commands.h"
#include "include/timer.h"
#include "include/use-time.h"

class greetings:
  def __init__(self, mouth):
    self.mouth = mouth

/* 始業挨拶 */
  def start_job(self, tim):
    self.mouth.ah()
    hour = tim.get_hour()
    if hour < MORNING_TIME:
      self.mouth.say(WORD_MORNING)
    elif hour > EVENING_TIME:
      self.mouth.say(WORD_EVENING)
    else:
      self.mouth.say(WORD_HELLO)
    tim.tell_date()
    self.mouth.ahah()


/* 単純な返事 */
  def yes (self):
    self.mouth.say_now(WORD_YES)

/* 業務終了挨拶 */
  def end_job (self):
    self.mouth.ah()
    self.mouth.say(WORD_SHUTTING)
    self.mouth.say_now(WORD_BYE)
```

検証

検証用スタブ test_greeting.py を用意します。

test_greeting.py

```
/* 挨拶モジュールの検証スタブ
   初版：2020/4/3 Chuji
   最新版：
*/

#include "include/use-time.h"

#include "time_keeper.py"
#include "mouth.py"
#include "greeting.py"

#define WAITING 5

fifo = open_FIFO()

mouth = speech_interface(fifo)
greet = greetings(mouth)
keeper = time_keeper(mouth)

greet.start_job(keeper)

sleep(WAITING)
greet.yes()

sleep(WAITING)
greet.end_job()
```

Redis への出力を確認するため、popper.py を二つ走らせて、REDIS_TO_MPC と REDIS_TO_WEB2 を監視させます。発話サブシステムを動作させていないので、音声合成結果は音声再生クライアント（REDIS_TO_MPC）に伝えられていません。

```
1:  $ cpp test_greeting.py |python
```

```
2:  $ python popper.py mpc & python popper Web2
    Text: Aah.wav from: mpc
    Text: こんにちは from: Web2
    Text: Ahello.wav from: mpc
    Text: 今日は 2020 年 4 月 3 日、金曜日です from:
    Web2
    Text: W from: mpc
    Text: Iahah.wav from: mpc
    Text: はい from: Web2
    Text: Iyes.wav from: mpc
```

```
Text: Aah.wav from: mpc
Text: システムをシャットダウンします from: Web2
Text: Ashutdown.wav from: mpc
Text: さようなら from: Web2
Text: Ibye.wav from: mpc
```

検証は、単に三つの操作を呼ぶだけです。greeting.py では SIMULATION を#define して time_keeper.py を呼び出す機能がありません。時刻による挨拶の変更を確認するためには、検証時の時刻に合わせて use-time.h の MORNING_TIME と EVENING_TIME を（仮に）調整してから検証をやり直し、挨拶が「おはようございます」→「こんにちは」→「こんばんは」と変わることを確認してください。

7.7.6命令解釈モジュール interpreter.py

カオナシへの命令を解釈して、各モジュールに業務の遂行指令を出すのが命令解釈モジュールです。プロトタイプでの命令と業務遂行は、以下の表のように設定しました。

代表的な命令	遂行する業務
（なし）	起動したら業務開始の挨拶をする
（命令のない呼びかけ）	挨拶を言う
シャットダウンしなさい	業務終了の挨拶をする システムをシャットダウンする
今日は何日？	今日の日付と曜日を言う
いま何時？	現在時刻を言う
5 分経ったら教えて	タイマーを起動し、起動したと言う タイマー時間を復唱する 時間が来たら、そう言う

命令解釈と業務の遂行

前の章で説明したように、if ～ then ～ else ～という構造を使って、命令の解釈と実行の規則を記述していきます。ここで大事になるのは、命令のインテンション（趣旨）をくみ取るということです。外国語を学んだことのある人は、主語と動詞と目的語さえ把握できれば、それほどトンチンカンでない会話ができたという経験があるでしょう。いまは命令に限っているので、主語はあまり重要ではありません。キーとなる動詞または目的語、あるいはその両方を見つけることができれば、インテンションを推測できます。命令文には揺らぎがある前提で、緩やかなパターンマッチングで処理します。

命令はウェイクアップワード「カオナシ」で始まるものにしますが、息継ぎのせいで、次の命令と別の文として認識されるかもしれません。そのときは「はい」とだけ答え、次の文を命令として処理する、簡単なステートマシンにしておきます。

設計

この節の始めに、命令や返事に使う日本語をインクルードファイル AI_commands.h に定義しました。もうひとつ、ステートマシンの動作に必要な定義もしておきます。

AI.h

```
/* AI 命令解釈モジュールで使う定義
   初版： 2020/3/23 Chuji
   最新版：
*/

#ifndef __AI_PARMS

/* ウェイクアップワードの効果 */
#define AI_SLEEPING  0
#define  AI_WAKEUP  1
#define AI_GOTO_SLEEP  3

/* ウェイクアップワードの削除 */
#define DEL_WAKEUP_WORD(x)  x[4:]

#define __AI_PARMS

#endif
```

命令解釈モジュールは以下のようなオブジェクトとして設計します。

クラス	interpreter	日本語テキストの解釈・実行
属性	wakeup	ウェイクアップワードの検出
	fifo	Redis FIFO オブジェクト
	tim	時間管理オブジェクト
	lip	発話インターフェースオブジェクト
	greet	挨拶オブジェクト
	emotion	感情表現オブジェクト
操作	interprete	日本語テキストの解釈と実行

interpreter.py のオブジェクト

これをもとに、各業務オブジェクトを生成し、左上の表に従って命令を解釈していきます。まだ評価段階なので、シャットダウン処理だけは禁止できるようにしてあります。

interpreter.py

```
/* AI 命令インタープリター
     初版： 2020/3/21 Chuji
     最新版：

Class: ai_interpreter - 日本語命令解釈
属性:
  wakeup:  ウェイクアップワード検出後の解釈テキスト数

  fifo:    Redis FIFO オブジェクト
  tim:     時間管理オブジェクト
  lip:     発話オブジェクト
  greet:   挨拶オブジェクト
  emotion: 感情表現オブジェクト

操作：
  interprete: 日本語テキストの解釈と実行

*/
```

```
#include "include/use-redis.h"
#include "include/AI_commands.h"
#include "include/AI.h"

#include "mouth.py"
#include "time_keeper.py"
#include "greeting.py"
#include "emotion.py"
#include "get_time.py"

class ai_interpreter:
  def __init__(self):
    self.wakeup = AI_SLEEPING

                /* オブジェクトの生成 */
    self.I_am = emotion()
    self.fifo = open_FIFO()
    self.lip = speech_interface(self.fifo)
    self.tim = time_keeper(self.lip)
    self.greet = greetings(self.lip)

    /* 業務開始の挨拶 */
    self.I_am.happy()
    self.greet.start_job(self.tim)

  def interprete(self, text):
    if NAME_KAONASHI in text:
      self.wakeup = AI_WAKEUP
      text = DEL_WAKEUP_WORD(text)

    if self.wakeup != AI_SLEEPING:

      if WORD_HELLO in text:
        self.I_am.fine()
        self.lip.repeat(text)
        self.lip.ah()
        self.lip.say(WORD_HELLO)
        self.lip.ahah()

      elif WORD_EVENING in text:
        self.I_am.serious()
        self.lip.repeat(text)
        self.lip.ah()
        self.lip.say(WORD_EVENING)
        self.lip.ahah()

      elif WORD_MORNIN in text:
        self.I_am.happy()
        self.lip.repeat(text)
        self.lip.ah()
        self.lip.say(WORD_MORNING)
        self.lip.ahah()

      elif ((WORD_TODAY in text) or (WORD_TODAYIS
in text)) and (WORD_WHATDAY in text):
            self.I_am.fine()
            self.lip.repeat(text)
            self.lip.ah()
            self.tim.tell_date()
            self.lip.ahah()

      elif ((WORD_NOW in text) or (WORD_TIMEIS in
text)) and (WORD_WHATTIME in text):
            self.I_am.fine()
            self.lip.repeat(text)
            self.lip.ah()
            self.tim.tell_time()
            self.lip.ahah()

      elif ((WORD_MINUTE in text) or (WORD_HOURS in
text)) and ((WORD_NOTIFY in text) or (WORD_TELLME
in text)):
        self.I_am.angry()
        self.lip.repeat(text)
        minutes = get_time(text)
        if minutes != 0:
          self.tim.run_timer(minutes)

      elif WORD_SHUTDOWN in text:
        self.I_am.angry()
        self.lip.repeat(text)
        self.greet.end_job()
#ifndef NO_SHUTDOWN
        subprocess.run(SYS_SHUTDOWN)
#endif
      else:
        text = ''
        if self.wakeup == AI_WAKEUP:
          self.I_am.serious()
          self.greet.yes()
        self.wakeup += 1
        if self.wakeup > AI_GOTO_SLEEP:
          self.wakeup = AI_SLEEPING

      return text
```

命令の解釈が決まったら、感情を表現し、ウェイクアップワードを除いた命令を復唱したら、所定の作業を実行します。この段階では、感情は単なる例として実装しており、特に意味はありません。

検証

検証では、日本語の命令を与えて、応答を調べることにします。

test_interpreter.py

```
/* test_interpreter.py 命令解釈モジュール検証
  初版：2020/4/4 Chuji
  最新版：
*/

#include "interpreter.py"

AI = ai_interpreter()

try:
  while True:
    text = input('日本語命令: ')
    AI.interprete(NAME_KAONASHI + text)
except KeyboardInterrupt:
  pass
```

この検証プログラムでは、第一の SSH 窓から日本語を手入力し、ウェイクアップワードを付け加えて interpreter に渡します。第二の SSH 窓で音声再生クライアント、Web サーバー、Open JTalk 各々の Redis キーに何が送られているか確認します。第二の SSH 窓のコピーの中に、青字で日本語命令を追記しました。

```
1:  $ cpp -DNO_SHUTDOWN test_interpreter.py
    >test.py
    $ python test.py
    EMOTION: I am happy.
    日本語命令: こんにちは
    EMOTION: I am fine.
    日本語命令: ああ
    EMOTION: I am concerned.
    日本語命令: おはよう
    EMOTION: I am happy.
    日本語命令: こんばんは
    EMOTION: I am sad.
    日本語命令: いま何時
    EMOTION: I am smiling.
    日本語命令: 今何時？
    EMOTION: I am smiling.
    日本語命令: 今日は何日？
    EMOTION: I am normal.
    日本語命令: きょうは何日
    EMOTION: I am normal.
    日本語命令: 5 分経ったら教えて
    EMOTION: I am interested.
    日本語命令: 十分後に知らせて
    EMOTION: I am interested.
```

```
日本語命令: シャットダウンして
EMOTION: I am smiling.
```

```
2:  $ python popper.py mpc & python popper Web1 &
python popper.py Web2 & python popper
OpenJTalk
Text: Aah.wav from: mpc          業務開始の挨拶
Text: こんにちは from: Web2
Text: Ahello.wav from: mpc
Text: 今日は 2020 年 4 月 4 日、土曜日です from:
Web2
Text: W from: mpc
Text: 今日は 2020 年 4 月 4 日、土曜日です from:
OpenJTalk
Text: Iahah.wav from: mpc
Text: こんにちは from: Web1
Text: Aah.wav from: mpc        「こんにちは」
Text: こんにちは from: Web2
Text: Ahello.wav from: mpc
Text: Iahah.wav from: mpc
Text: はい from: Web2     意味のない「ああ」への返事
Text: Iyes.wav from: mpc
Text: おはよう from: Web1          「おはよう」
Text: Aah.wav from: mpc
Text: おはようございます from: Web2
Text: Agood_morning.wav from: mpc
Text: Iahah.wav from: mpc
Text: こんばんは from: Web1        「こんばんは」
Text: Aah.wav from: mpc
Text: こんばんは from: Web2
Text: Agood_evening.wav from: mpc
Text: Iahah.wav from: mpc
Text: いま何時 from: Web1              「いま何時」
Text: Aah.wav from: mpc
Text: 今、午後 5 時 26 分です from: Web2
Text: 今、午後 5 時 26 分です from: OpenJTalk
Text: W from: mpc
Text: Iahah.wav from: mpc
Text: 今何時？ from: Web1              「今何時？」
Text: Aah.wav from: mpc
Text: 今、午後 5 時 26 分です from: Web2
Text: 今、午後 5 時 26 分です from: OpenJTalk
Text: W from: mpc
Text: Iahah.wav from: mpc
Text: 今日は何日？ from: Web1       「今日は何日？」
Text: Aah.wav from: mpc
Text: 今日は 2020 年 4 月 4 日、土曜日です from:
Web2
Text: 今日は 2020 年 4 月 4 日、土曜日です from:
OpenJTalk
Text: W from: mpc
Text: Iahah.wav from: mpc
Text: きょうは何日 from: Web1      「きょうは何日」
Text: Aah.wav from: mpc
Text: 今日は 2020 年 4 月 4 日、土曜日です from:
Web2
Text: 今日は 2020 年 4 月 4 日、土曜日です from:
OpenJTalk
Text: W from: mpc
Text: Iahah.wav from: mpc
Text: 5 分経ったら教えて from: Web1 「5 分経ったら
教えて」
Text: Aah.wav from: mpc
Text: 1 番タイマーを起動しました。 from: Web2
Text: Atimer_start_1.wav from: mpc
Text: 5 分後にお知らせします。 from: Web2
Text: W from: mpc
Text: 5 分後にお知らせします。 from: OpenJTalk
Text: Iahah.wav from: mpc
Text: 十分後に知らせて from: Web1  「十分後に知らせ
て」
Text: Aah.wav from: mpc
Text: 2 番タイマーを起動しました。 from: Web2
Text: Atimer_start_2.wav from: mpc
Text: 10 分後にお知らせします。 from: Web2
Text: W from: mpc
Text: 10 分後にお知らせします。 from: OpenJTalk
Text: Iahah.wav from: mpc
Text: シャットダウンして from: Web1「シャットダウン
して」
Text: システムをシャットダウンします from: Web2
```

```
Text: Aah.wav from: mpc
Text: さようなら from: Web2
Text: Ashutdown.wav from: mpc
Text: Ibye.wav from: mpc
```

繰り返して検証できるよう、ファイル入力ができる検証プログラムを用意しました。ウェイクアップワードは検証プログラムでは追加せず、入力ファイル test_interpreter2.txt の中の日本語の先頭に加えています。ウェイクアップワードの後ろに何もない場合、次の二つの文までは処理することを確認する入力を加えています。結果は掲載しませんが、test_interpreter2.txt の内容から予想してみてください。

test_interpreter2.py

```
/* test_interpreter2.py 命令解釈モジュール検証--ファイル
から入力
  初版：2020/4/4 Chuji
  最新版：
*/

#include "interpreter.py"

AI = ai_interpreter()

try:
  while True:
    text = input()
    print (text)
    AI.interprete(text)
except KeyboardInterrupt:
  pass
```

test_interpreter.txt

```
カオナシこんにちは
カオナシおはよう
カオナシこんばんは
カオナシいま何時
カオナシ今何時？
カオナシ今日は何日？
カオナシきょうは何日
カオナシ 5 分経ったら教えて
カオナシ十分後に知らせて
カオナシシャットダウンして
カオナシああ
こんにちは
おはよう
こんばんは
```

7.7.7AI エンジン冒頭部　kaonashi.py

最後は AI エンジンサブシステムの冒頭部（本体）である kaonashi.py を設計します。

設計

検証やデバグ用に以下のマクロを使っています。AI_DEBUG は下位のモジュール interpreter.py を検証用スタブ interpreter_stub.py に置き換えます。RUN_ALL を#define すれば、他のサブシステム（聴覚、発話、Web サーバー）をすべて起動します。

マクロ	#ifdef	#ifndef
AI_DEBUG	interpreter にスタブを使う	interpreter.py を使う
RUN_ALL	サブシステムを起動する	サブシステムは起動しない
DEBUG	日本語命令文を表示する	表示しない

kaonashi.py のマクロ定義

kaonashi.py の機能は単純で、聴覚サブシステムと発話サブシステムを起動したら、REDIS_TO_AI から受け取った日本語テキストを命令解釈モジュールに渡すだけです。

kaonashi.py

```
/* 秘書ロボットカオナシ

   初版：  2020/3/28  Chuji
   最新版：
*/

#include "include/gpio.h"
#include "include/use-sys.h"
#include "include/use-redis.h"

#ifdef AI_DEBUG
#include "interpreter_stub.py"
#else
#include "interpreter.py"
#endif

#define RUN_EAR  './run_ear'
#define RUN_VOICE  './run_voice'
#define RUN_WEB  './run_web'

/* 常駐プロセスの起動 */

#ifdef RUN_ALL
sfork(RUN_EAR)
sfork(RUN_WEB)
sfork(RUN_VOICE)
#endif

/* 子オブジェクトの生成 */

fifo = open_FIFO()
AI = ai_interpreter()

try:
  while True: /* 割り込み待ちに入る手順 */
    tag, text = fifo.blpop(REDIS_TO_AI)
#if DEBUG
    print('AI received: ', text.decode())
#else
    AI.interprete(text.decode())
#endif

except KeyboardInterrupt:
#ifdef BCM2835
  GPIO.cleanup()
#endif
  pass
```

検証

検証スタブ interpreter_stub.py は、渡された日本語テキストを表示するだけです。つまり AI_DEBUG を#define したときは、test_ear.py と同じ動作をすることになります。

interpreter_stub.py

```
/* AI 命令インタープリターの代わりをするスタブ
     初版：  2020/4/5 Chuji
     最新版：

Class: ai_interpreter - 日本語命令解釈
属性:
  なし
操作：
  interprete: 日本語テキストの解釈と実行

*/

#include "include/use-redis.h"
#include "include/AI_commands.h"
#include "include/AI.h"


class ai_interpreter:
  def __init__(self):
    pass

  def interprete(self, text):
    print('Text to interprete: ', text)
```

検証のため AI_DEBUG を#define して、Redis pusher から任意のテキストを送ります。この段階では日本語の解釈を行っていないので、どんなテキストでも伝わることが分かります。

1:
```
$ python pusher.py AI
Hello World!
Pushed: Hello World!
カオナシ、こんにちは
Pushed: カオナシ、こんにちは
```

2:
```
$ cpp -DAI_DEBUG kaonashi.py | python
Text to interprete:  Hello World!
Text to interprete:  カオナシ、こんにちは
```

AI エンジンの単体評価はこれで終了です。次は次章で、サブシステムを組み合わせた動作を評価していきます。

コラム　日本のロボット史（その 2）

日本のロボット史を飾る、代表的なコミックとテレビ番組を挙げてみます。何作ご存じますか？あなたのお好みは？

1955　ロボット三等兵（前谷惟光）
1963　鉄腕アトム（手塚治虫）
1963　鉄人 28 号（横山光輝）
1966　ロボタン（森田拳次）
1972　マジンガーZ（永井豪）
1974　がんばれロボコン（石ノ森章太郎）
1979　機動戦士ガンダム（矢立肇、富野喜幸）
1979　バトルフィーバー（スーパー戦隊と巨大ロボ初作）
1981　ロボット 8 ちゃん（石ノ森章太郎）

8　プロトタイプの評価と実証モデルに向けての検討

前章までに、秘書ロボット「カオナシ」フェーズ 3（プロトタイプ）の設計と、サブシステム単位のコーディングと検証を行いました。この章では、秘書ロボットとしての総合的な評価を行い、フェーズ 4（実証モデル）の開発に向けた課題を検討します。

8.1　総合検証

それではカオナシを起動して、動作を確認してみましょう。

8.1.1サブシステムの立ち上げ

最初は、動作を確認しながらサブシステムを立ち上げてみます。聴覚サブシステムは CPU 時間を食うので、しばらくの間は使わずに（RUN_ALL は #define しないで）進めることにします。その代わり、Web サーバーと発話サブシステムを手動で起動します。

```
$ ./run_web & sudo ./run_voice &
$ cpp kaonashi.py | python
```

カオナシを起動すると、音声で始業挨拶が聞こえてきます。

8.1.2ブラウザからの命令

ブラウザで「カオナシ Web」サイト（私の環境では http://192.168.15.16:50010/）にアクセスすると、ロボットの応答欄に、（タイミングによっては）始業挨拶の最後の言葉が残っていることがあります。

何か命令を与えてみましょう。とりあえず「おはよう」と入力して、送信ボタンをクリックすると、すぐに「おはようございます」という返事が表示されます。それから「アッ、おはようございます、アッアー」という声が聞こえてきました。「アッ」と「アッアー」は、それぞれ返事の最初と最後を示すために発声しています。

プロトタイプで定義した命令を与えると、すべてに設計どおりの返事が返ってきました。時刻や日付を訊くと、ブラウザにはすぐ返事が表示されますが、合成した音声が聞こえるのは 2〜3 秒後です。

8.1.3音声命令

今度は音声で命令を与えます。常駐させた Web サーバー、Julius、音声再生クライアントは kill -9 で終了させておくか、面倒ならリブートしてから全システムを起動します。

```
$ cpp -DRUN_ALL -DBCM2835 kaonashi.py | sudo python
```

すべてのプロセスが立ち上がり、音声命令を受け付けられるようになるまで 30 秒ほど待ってから、命令を言ってみましょう。最初のうちは音声認識がうまく働かないので、何度も声がけをしてください。やがて、命令の長さにもよりますが、10~30 秒後に返事が返ってくるようになります。

8.2　パフォーマンス評価

プロトタイプの語彙はあまり多くありませんが、その範囲で返事が返ってくるまでの時間を計ってみます。命令を言い終えてから、Web に返事が表示されるまでの時間と、音声が発声されるまでの時間を下表に示します。定型の応答は、ほぼ同時に発声されます。音声合成をする場合は、Web の表示から 3〜5 秒後に返事が発声されました。

音声命令	Web に表示するまで	音声まで
カオナシ	5〜10 秒	←
カオナシ、こんにちは	30 秒	←
カオナシ、いま何時ですか	25〜30 秒	+5 秒
カオナシ、今日は何日ですか	30 秒	+5 秒
カオナシ、三分たったら教えて	30 秒	+5 秒

プロトタイプが応答を返すまでの時間

Julius の音声認識時間が、単体で動かしたときの 2 倍になってしまいました。マルチプロセス化の影響やクロックを変更したことで、Julius が CPU を使える時間が半減したということでしょう。このくらい時間がかかると、果たして命令が認識されたのか分からず、不安になってしまいそうです。

窓から外の騒音が入ってくると、それも録音して処理しようとします。これが続くと、音声命令の認識に取りかかるのが遅くなってしまいます。

時間がかかるようになった要因のもう一つは、長い命令を与えるようになったことです。「いま何時？」は、なかなか正確に認識できません。「いま何時ですか？」と言うと、認識率が良くなります。命令を文として認識しているせいです。

ウェイクアップワードの「カオナシ」は捕まえても、そのあとの命令を認識できないと、むなしく「はい」という返事だけが返ってきます。助走期間をとるのと、雑音を減らして明瞭に発音する必要があるようです。

8.3　実証モデルにむけた改善項目

プロトタイプを動作させてみると、特に次の三項目を改善したくなります。

1. 音声認識の処理時間を短縮したい
2. 感情表現をもっと豊かにしたい
3. カオナシ Web にロボットとの会話履歴を表示したい

実証モデルで実現を目指しますが、その方向を検討しておきましょう。

8.3.1音声認識の処理時間短縮

音声認識 Julius は、やっぱり Raspberry Pi ZERO には荷が重いようです。というより、命令を与える方にとって、我慢ならないくらい時間がかかっています。マルチプロセス化したことで、(他のプロセスも同時にこなしているので) デモモードの時よりさらに遅くなってしまったようです。この処理時間を短縮するためには、別のサーバーの力を借りなければなりません。選択肢としては、

A. ネットワーク上にある、もっと CPU パワーのあるサーバーを利用する
B. Julius をあきらめて、クラウドサービスを利用する

が考えられます。ここまで Julius を試したことだし、まずは選択肢 A で進めてみます。私の家には、常時運転しているサーバーが一台だけあります。Ubuntu が走っている古いネットブックですが、半導体ディスクにして、それなりに便利に使っています。サーバーの負荷は、あまり高くないので、アルバイトをすることは可能です。

まず、フェーズ 2 で行った手順で、このサーバーに Julius をインストールしたら、音声をネットワーク経由で受信するモードで起動します。

```
$ julius -C /home/chuji/dictation-kit-4.5/main.jconf -C /home/chuji/dictation-kit-4.5/am-gmm.jconf -input adinnet -realtime -cutsilence -module
```

サーバーで Julius を起動する

音声は adinnet（TCP ポート番号 5530）というプロトコルを使っています。ここで使っているオプションは以下のとおりです。

オプション	意味
-C	設定ファイルの指定（フェーズ 2 と同じ）
-input	音声入力の指定（ネットワーク経由）
-realtime	音声を受け取りながら解析を進める
-cutsilence	無音区間で認識範囲を分離する
-module	モジュールモードで動かす（フェーズ 3 と同じ）

サーバーでの Julius オプション

Raspberry Pi ZERO では、音声をデジタル信号にして、サーバーに送ってやります。そのための adintool というプログラムは、Julius の一部として既にインストールされています。

しかし、その前に Julius の認識結果を受け取るクライアントを起動しておく必要があります。Julius は出力先がないうちは、音声信号の受け取りを拒否してしまうからです。この説明がある資料がなかなか見つからず、ちょっと苦労しました。クライアントとしては、ear.py がそのまま使えます。インクルードファイル tcp.h で接続先 IP アドレスを自局内（127.0.0.0）から、サーバーのそれに変更し、マクロ名 REMOTE_JULIUS を#define すえば OK です。

tcp.h（改造箇所のみを赤字で示す）

```
 :
 #ifdef REMOTE_JULIUS
 #define TCP_LOOPBACK  '192……' /* サーバーの IP アドレスをここに入れる */
 #else
 #define TCP_LOOPBACK '127.0.0.1'
 #endif
 :
```

機能を確認するため、Julius の XML 出力と認識結果を表示するように起動します。3 つのソフトウエアを立ち上げる順番を、もう一度整理しておきます。

1. まずサーバーで Julius（音声はネットワーク入力）を立ち上げておく
2. Raspberry Pi ZERO でクライアント ear.py を起動し、Julius に接続して認識結果を受け取るようにする
3. Raspberry Pi ZERO で adintool を起動し、Julius に接続して音声信号を送れるようにする

まず第 1 の SSH 窓でクライアント ear.py を立ち上げてから、次に第 2 の SSH 窓で adintool を起動します。下の第 2 の SSH 窓では、サーバーの IP アドレスの一部を削除しているので、皆さんのサーバーに合わせてください。

```
1: $ cpp -DTEST_LINE -DREMOTE_JULIUS -DHEARING_TEST ear.py | python
2: $ sudo adintool -in mic -out adinnet -server 192.168.… -freq 16000
```

Julius クライアントと音声入力を起動する

音声入力 adintool のオプションは次のとおりです。

オプション	意味
-in	音声入力デバイス（mic）の指定
-out	デジタル音声の出力先（adinnet）を指定
-server	サーバーの IP アドレスを指定
-freq	デジタル化周波数（デフォルト 16kHz）を指定

サーバーでの Julius オプション

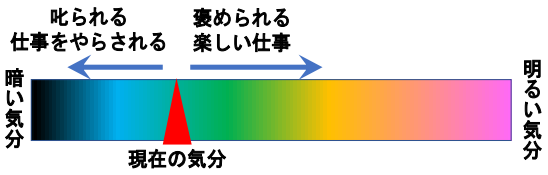
ロボットに話しかけると、サーバーで認識する経過が第 1 の SSH 窓に表示されます。大まかに言って 3~5 秒で認識できているので、大きな改善になることが分かりました。もっと CPU パワーのある PC が動いているときは、こちらを使うように設定できれば、さらに改善が望めます。

Julius をサーバーに常駐させるメリットがもう一つあります。起動したばかりの Julius は認識精度が低く、音声入力を与えていくと動作パラメータを自動で調整していきます。ところがロボットの電源を切ると、すべてを忘れてしまい、また最初からやり直すことになります。サーバーに常駐させておけば、この助走期間が短縮できると期待しています。

8.3.2感情表現の改善

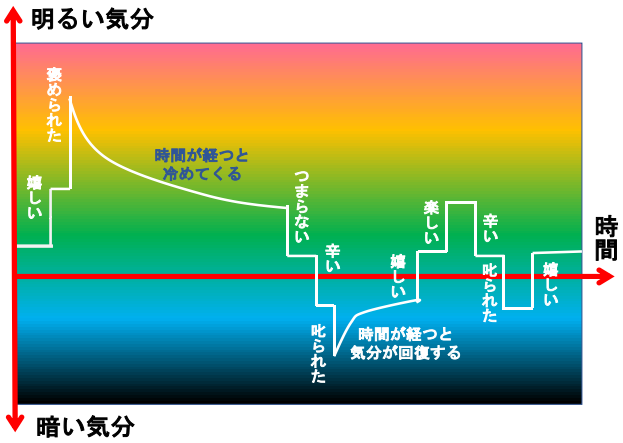
プロトタイプの感情表現は、命令ごとに「表情が変わった」ことを示すだけでした。ハードウエアの制御を確立するのが主目的だったので、表現では手を抜いたのが原因です。

ヒトの感情というのは（たぶん妖怪も同じで）、過去の出来事を引きずるいっぽう、時がたつと高揚感も冷めていくという、時間に依存する現象です。この単純なモデルとして、下図のような「気分変数」で表現してみます。現在の気分は、褒められれば明るい方に移るし、仕事がつらいと暗くなっていきます。



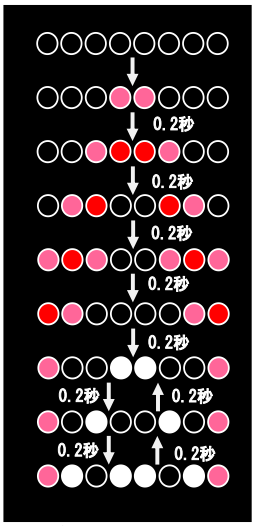
気分変数

「気分変数」は右上の図のような時間変化をするという前提で設計してみます。業務がうまく行ったり、褒められたりすれば、それに応じてプラス側（明るい側）に変化し、きつい仕事だったり、叱責されたりすれば、マイナス側に変化します。ただし、ある程度プラスあるいはマイナス側に行き過ぎた時は、時間が経つと平静な値（ゼロではない）に近づいてきます。ステップ変化の幅や、変化率はパラメータとして、人間側が楽しめる値に調節することにします。



気分変数の時間変化をモデル化する

気分変数の値によって、カオナシの「感情」を変化させます。声色を変えようとすると、すべての発話音声を合成することになり、応答が悪くなりそうなので、しばらくの間はあきらめます。その代わり、カラーLED の発光パターンを変化させて、表情を模擬することにします。例えば「怒りの表情」では、カオナシが大口をあけて、毒気を吹き出すイメージを表現してみます（ゴジラが口から炎を吐くと言った方が分かりやすい？）。表現の時間変化を右図のような「シナリオ」に仕上げたら、コーディングに移れます。ただし、このシナリオの良しあしはセンスで決まるし、私のシナリオが良いものであるという自信もないので、シナリオ設計の体系的な説明はしません。

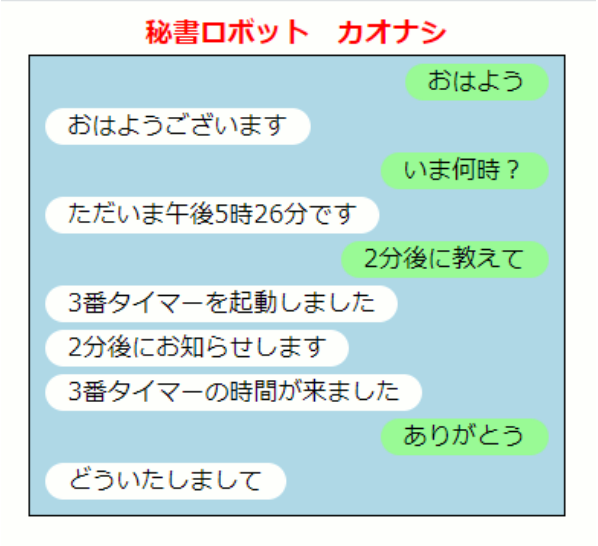


怒りのシナリオ

この表現手法が、ソフトウエアの構造に与える影響は見逃せません。過去の感情を引きずるためには、内部変数（オブジェクトモデルの属性）が必要だし、他の業務遂行と並行して発光パターンを変化させるため、プロセスとして設計することになります。これは実証モデルで実現させます。

8.3.3カオナシ Web 画面の会話履歴

プロトタイプの Web イメージは、AI エンジンの検証に使えるというのが最低要件だったので、あまりにも貧弱です。少なくとも、数回分の命令と応答の履歴は表示して欲しいと思います。SNS のイメージを意識して、下のような画面を設計してみました。



ここで検討が必要なのが、対話の履歴をどこに残すかという点です。SNS ではサーバーに保存して、過去の履歴も見ることができるようになっています。しかしここでは、あえてブラウザが覚えることにし、サーバーは以前のものが使えるようにしました。秘書ロボットの動作を確認するという目的であれば、ブラウザが接続してからの履歴で充分だからです。アイコンや発言時刻の表示もしていません。

最大表示数は 10 行とし、それを越えたときは古い（一番上に表示されている）ものを消していきます。改造箇所は次のようになりました。

以下に示すファイルの赤字で示した部分が、このための変更箇所です。青字で示しているのは、画像を表示するための変更箇所で、この後で説明します。

Web_if.h（改造箇所のみ赤字と青字で示す）

```
/* ロボットの Web インターフェース定義
   初版： 2020/3/19 Chuji
   最新版：2020/4/11 対話領域増設、カメラ表示追加
*/
 （中略）
/* html ページのデザイン */

#define WEB_TITLE_FONT  size = "4" color="red"
#define WEB_TITLE  秘書ロボット　カオナシ

/* スタイルシートの名前 */
#define NAME_FRAME  frame
#define NAME_ROBOT  robot
#define NAME_MASTER  master
#define NAME_BLANK  blank

#define ID_FRAME  'frame'
#define ID_ROBOT  'robot'
#define ID_MASTER  'master'
#define ID_BLANK  'blank'

/* スタイルシート */
#define STYLE_FRAME  .NAME_FRAME{width: 350px;
height: 300px; border: solid black thin;
background-color: lightblue; bottom:
20px;}.frame{width: 350px; height: 300px; border:
solid black thin; background-color: lightblue;
bottom: 20px;}

#define STYLE_ROBOT  .NAME_ROBOT{background-color:
white;text-align: left; color: black; border-
radius:15px; margin-top: 5px; margin-left: 10px;
margin-right: auto; width: fit-content; padding: 0
15px;}

#define STYLE_MASTER  .NAME_MASTER{background-
color: palegreen; text-align: right; color: black;
border-radius:15px; margin-top: 5px; margin-right:
10px; margin-left: auto; width: fit-content;
padding: 0 15px;}
#define STYLE_BLANK  .NAME_BLANK{background-color:
lightblue;}

/* プロパティのコピーと設定 */
#define COPY_ATTR(y,x,attr)
document.getElementById(y).attr=document.getElement
ById(x).attr
#define SET_ATTR(y,x,attr)
document.getElementById(y).attr=x

/* 対話領域設定 */
#define CHAT_LINE(x)  <div id=x class=ID_BLANK>
</div>

/* 表示・入力領域定義 */
#define WEB_ID_COMMAND  IO_EVENT_COMMAND
#define WEB_ID_ROBOT  IO_EVENT_ROBOT
#define WEB_ID_MASTER  IO_EVENT_MASTER
#define WEB_ID_IMAGE  IO_EVENT_CAMERA

 （中略）
/* カメラ画像表示領域 */
#define CAMERA_IMAGE  'camera image'
#define DEFAULT_IMAGE  'kaonashi.jpg'
#define DISPLAY_IMAGE_SIZE  '640'

/* 対話領域定義 */
#define CHATS  10
#define CHAT  'chat'
#define CHAT1  'chat1'
#define CHAT2  'chat2'
#define CHAT3  'chat3'
#define CHAT4  'chat4'
#define CHAT5  'chat5'
#define CHAT6  'chat6'
#define CHAT7  'chat7'
#define CHAT8  'chat8'
#define CHAT9  'chat9'
#define CHAT10  'chat10'

#define __WEB_IF
#endif
```

最初の追加部分は、表示部の修飾を定義するスタイルシート（CSS）です。表示部の形や色、位置取りなどを SNS のイメージに近づけるように指定しています。このうち、囲い枠の大きさを内容に合わせて変更する width:fit-content という値は、一部のブラウザでは機能しません（囲い枠が幅いっぱいになる）。しかし、私の Chrome では使えるし、機能しなくても致命的ではないので採用しました。ブラウザの種類を調べ、それに合わせて値の名前を変更すれば同じ効果が得られる（IE を除く）のですが、面倒なので止めました。CSS の詳しい説明は省

略するので、興味のある人は他の文献にあたってください。

次のマクロ定義は、下の行の内容をすぐ上の行にコピーする手続きと、一番下の行を設定する手続きです。本文に埋め込むと、長くなって趣旨を見失いやすいので、見やすくする工夫になっています。

最後の追加は、表示領域に関する定義です。

HTML 文書のソースは以下のようになりました。

index.html.source（改造箇所のみ赤字と青字で示す）

```
/* ブラウザからの操作用 index.html ファイルのソース
    初版；2020/3/19 Chuji
    最新版：2020/4/13 -- 対話領域増設、画像表示領域新設

    注：使用する前に cpp で処理すること
    $ cpp index.html.source | python cleanfile.py
>index.html
*/
 （中略）

<script SCRIPT_SOCKETIO></script>

<style>
  STYLE_FRAME
  STYLE_MASTER
  STYLE_ROBOT
  STYLE_BLANK
</style>

<script type="text/javascript">

 （中略）

/* 表示要求の処理 */

/* 対話領域のスクロール */
function scroll(msg){
  for (var i=CHATS; i>1; i--){
    COPY_ATTR(CHAT+String(i),CHAT+String(i-
1),textContent);
    COPY_ATTR(CHAT+String(i),CHAT+String(i-
1),className);
  }
  SET_ATTR(CHAT1,msg,textContent);
}

/* マスターのコマンド */
mysock.on(IO_EVENT_MASTER, function(msg){
#ifdef DEBUG
  obj=document.getElementById(WEB_MESSAGE_RAW);
  obj.textContent="Master's words are received:
"+msg;
#endif
  scroll(msg);
  SET_ATTR(CHAT1,ID_MASTER,className);
});

/* ロボットの応答 */
mysock.on(IO_EVENT_ROBOT, function(msg){
#ifdef DEBUG
  obj=document.getElementById(WEB_MESSAGE_RAW);
  obj.textContent="Robot's words are received:
"+msg;
#endif
  scroll(msg);
  SET_ATTR(CHAT1,ID_ROBOT,className);
});

/* 画像の更新 */
var counter=0;
mysock.on(IO_EVENT_CAMERA, function(msg){
#ifdef DEBUG
  obj=document.getElementById(WEB_MESSAGE_RAW);
  obj.textContent="Robot's words are received:
"+msg;
#endif
SET_ATTR(WEB_ID_IMAGE,msg+'?'+String(counter),src);
  counter++;
});

 （中略）
/* HTML 主要部 */
<center>
<font WEB_TITLE_FONT><b>WEB_TITLE</b></font>
</center>

#ifdef DEBUG
<p><div id=WEB_MESSAGE_RAW>サーバーからの受信データ
</div></p>
<p><div id=WEB_MESSAGE_RAW2>ブラウザからの送信データ
</div></p>
#endif

/* 画像表示領域 */
<center>
<img id=WEB_ID_IMAGE src=DEFAULT_IMAGE
width=DISPLAY_IMAGE_SIZE>
</center>
<br>

/* 会話表示領域 */
<center>
<div class=ID_FRAME>
  CHAT_LINE(CHAT10)
  CHAT_LINE(CHAT9)
  CHAT_LINE(CHAT8)
  CHAT_LINE(CHAT7)
  CHAT_LINE(CHAT6)
  CHAT_LINE(CHAT5)
  CHAT_LINE(CHAT4)
  CHAT_LINE(CHAT3)
  CHAT_LINE(CHAT2)
  CHAT_LINE(CHAT1)
</div>
</center>

 （中略）
</body>
</html>
```

冒頭にはスタイルシートを置いています。次の関数 scroll は、表示行のスクロールを実行するため、上から順番にひとつ下の表示行の内容とスタイルをコピーします。関数の最後で、受信したテキストを一番下の行に表示します。Socket.IO のハンドラでは、これを呼び出してから、該当するスタイルシートを適用しています。

最後に会話を表示する領域を HTML で定義しています。行数 CHATS を使って自動生成することもできるのですが、ここでは固定枠として記述し、外枠に収まるように調整しています。

プロトタイプと差し替えて動作させると、だいぶ見やすくなったことが分かります。前に述べた理由で、ブラウザは Chrome がおすすめです。

8.4　実証モデルで追加する機能の検討

プロトタイプの動作を見て、フェーズ 4 実証モデルで追加する機能と実現手段を検討しておきます。

8.4.1挨拶の充実

秘書ロボットとの会話が自然なものになるよう、簡単な挨拶をさらに充実させます。実はプロトタイプでは、当初設計にない会話をすでに組み込んでいました（下表の上側にある 3 項目）。

代表的な語りかけ	返事
おはよう（プロトタイプで実装済）	おはようございます
こんにちは（プロトタイプで実装済）	こんにちは
こんばんは（プロトタイプで実装済）	こんばんは
おやすみ（シャットダウンさせる）	おやすみなさい
さようなら（シャットダウンさせる）	さようなら
ありがとう	どういたしまして
よくやった（または「ごくろうさん」）	ありがとうございます
いい天気ですね	そうですね
お元気ですか	おかげさまで元気です
起きて（または「ねえねえ」）	はい
疲れた（または「眠い」）	元気を出してください

追加する挨拶

8.4.2画像取得

カメラ画像を取り込んで Python で処理するためのライブラリをインストールします。標準ディストリビューションに含まれているという噂があったので、インポートできるか試してみましたが、できなかったのでインストールしました。こんどはインポートできます。

```
$ python -c "import picamera"
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ModuleNotFoundError: No module named 'picamera'

$ sudo apt-get install python3-picamera
 :
$  python -c "import picamera"
$
```

このライブラリの使いかたについては、説明ページ（https://picamera.readthedocs.io/en/release-1.13/）を参照してください。

これを使って、カメラで撮影した画像を（ファイル名 image1.jpg～image5.jpg を使いまわしながら）、Redis キーREDIS_TO_DISP へ送る試験プログラムを作りました。画像ファイルは Web ページと同じディレクトリ（home/chuji/kaonashi）に置きます。負荷を抑えるため、次の撮影まで 1 秒待つようにしました。

test_camera.py

```
/* カメラ動作試験
     初版：2020/4/13 Chuji
     最新版：
*/

#include "include/use-redis.h"

from time import sleep
from picamera import PiCamera

camera = PiCamera()
camera.resolution = (640,480)

image = 1
fifo = open_FIFO()

try:
  while True:
    file_name = 'image'+str(image)+'.jpg'
    camera.capture(file_name)
    fifo.rpush(REDIS_TO_DISP, file_name)
    image += 1
    if image > 5:
      image = 1
    sleep(1)
except KeyboardInterrupt:
  pass
```

次の画像表示と合わせて検証できます。

8.4.3画像表示

撮影した画像をブラウザに表示できるようにしましょう。サーバーはあまり変わりません。新しい Redis キーREDIS_TO_DISP にファイル名が渡ってくるので、それを Socket.IO にメッセージ付きイベントとして転送する部分を追加しました。ブラウザから要求された URL のうち、'?'より後ろは、すぐ下の説明にあるように、切り捨ててファイルを探します。

web-if.js（改造箇所のみ赤字で示す）

```
/* 秘書ロボット用 Web サーバー
   オリジナル：温度コントローラ  2017/6/5-2019//7/10
   初版：2020/3/19
   最新版：2020/4/13 ---カメラ画像更新情報追加
*/
（中略）
  var filepath = CURRENT_DIR + request.url;
  var pos=filepath.indexOf('?');
  if (pos != -1){
    filepath=filepath.substr(0,pos);
  }
  if (filepath == NO_EXPLICIT_FILE) {
      filepath= filepath + INDEX_HTML;
  }
  fs.readFile(filepath, function(error,
filecontent) {
    if (!error){
      response.writeHead(JS_HTTP_STATUS_OK,
JS_HTTP_HEADER(getType(filepath)));
      response.end(filecontent, JS_CHAR_UTF8);
    }
  });
}).listen(PORT_SOCKET_IO);

（中略）

/* Redis を介した AI エンジンとのインターフェース */

var in_queue1 =
require(JS_REQ_REDIS).createClient();
var in_queue2 =
require(JS_REQ_REDIS).createClient();
var in_queue3 =
require(JS_REQ_REDIS).createClient();
```

```
var out_queue =
require(JS_REQ_REDIS).createClient();

in_queue1.flushdb();

in_queue1.blpop(REDIS_TO_WEB1, REDIS_NO_TIMEOUT,
show_master);
in_queue2.blpop(REDIS_TO_WEB2, REDIS_NO_TIMEOUT,
show_robot);
in_queue3.blpop(REDIS_TO_DISP, REDIS_NO_TIMEOUT,
show_image);

 (中略)

function show_image(err, msg){
  show(IO_EVENT_CAMERA, msg[REDIS_DATA]);
  in_queue3.blpop(REDIS_TO_DISP, REDIS_NO_TIMEOUT,
show_image);
 }
```

HTML 文書の改造箇所は、この前に会話履歴を付け加えたときのファイル（青字部分）で示してあります。インクルードファイル web_if.h についても同様です。

まず新しいイベントの処理を追加します。この部分は、すぐ前に会話履歴を表示したのと同じマクロが使えます。使いまわしたファイル名をブラウザに表示させると、前に取得したファイル（キャッシュに入っている）を表示しようとします。ファイル名の後ろにパラメータ?xxx と付けるのは、キャッシュからの読み込みを禁止するためです。パラメータはサーバー内で除去します。

画像表示領域には、最初はカオナシの画像（配布ファイルには含まれていません）を表示させておきますが、新しいファイル名が Socket.IO を介して伝わってきたら、この ID を使って更新します。画像のサイズが変わっても良いように、width を指定しておきます。

Web サーバーと HTML 文書 index.html を改造版と交換し、Web サーバーを起動します。

test_camera.py を走らせると、ブラウザ上の表示画像がコマ送りのように更新されます。これでカメラの向きやピントが調整できるようになりました。

8.4.4人物認証、名前管理、人物情報

当初の計画では、「カオナシ、こんにちは」と語りかけたら、相手の顔を見て、「〇〇さん、こんにちは」と返事することを考えていました。クラウドを使って実現できるのですが、学習用の写真を集めるのが面倒なのと、認証すべき人物の管理と登録はもっと面倒なので、後回しにすることにしました。

その代わり、人物（あるいはグループ）の写真を撮って、メールする機能をつけようと思います。フルモデルでは口述した文面をメールする目標を立てているので、その準備として送信機能を実装します。

8.4.5調査・検索、天気照会

ロボット本体にない情報は、インターネット経由で入手しなければなりません。これには、開発環境が公開されているクラウドサービスを利用しようと思います。つまり、スマートスピーカーになるということです。ここで問題になるのは、現在の音声コマンド処理では、クラウドに接続するのが、そのコマンドを処理した後になるということです。たとえば「今日の天気は？」という問いかけを認識したとすると、その音声データは Julius が使ってしまって、どこにも残っていません。考えられる方法は以下のとおりです。

A. Julius が認識した日本語テキストを音声データに戻してからクラウドに送りつける
B. Julius が認識した日本語テキストをクラウドに送りつける
C. カオナシへ命令して、クラウドに接続させ、あらためてクラウドの命令を言う

一見したところ、選択肢 C が自然そうですが、「カオナシ、アレクサに繋いで」→「アレクサに繋ぎました」→「アレクサ、今日の天気は？」という会話になり、冗長な感じがします。それより選択肢 B がスマートだと思います。Google Assistant では、要求をテキストで送りつける方法が公開されています。Alexa でも、音声認識（ASR）をバイパスして、直接自然言語理解（NLU）に送りつけることができそうですが、まだ（サービスが公開されているかも含めて）良い資料が見つからず、すぐには使えません。

代表的な語りかけ	返事
今日の天気予報は？	アレクサがお答えします （アレクサの返事が続く）
近くの美味しいレストランを教えて	

実証モデルでの検索項目（Alexa の場合）

8.4.6音楽などの mp3 ファイルの再生

プロトタイプで扱った音声ファイルは、すべて wav 形式でした。ところがクラウドサービスが返す音声は、圧縮が可能な形式、多くは mp3 形式です。aplay が再生できるのは wav ファイルだけで、mp3 ファイルの再生には別コマンド mpg123 が必要です。両者を別々のプロセスとして同時に使うと、再生のタイミング調整が複雑になりすぎます。

異なる音声ファイル形式を使い分けられる、既存のライブラリを調べ、試してみました。pygame が実例としてよく引用されますが、音声ファイルのサンプリングレート（周波数）を読み取って再生することができませんでした。名前から分かるように、python でゲームを設計するためのパッケージで、その必要がなかったのでしょう。一番安定していたのがビデオ再生にも使える VLC というライブラリでした。主な操作を下表に示します。

vlc の操作	操作内容
`MediaPlayer()`	モジュールの初期化
`set_mrl(`*`file`*`)`	音声ファイルの再生を準備する
`play()`	音声ファイルを再生する
`audio_get_volume()`	音声再生の設定音量(0～100)を知る
`audio_set_volume(`*`vol`*`)`	音声再生の音量(0～100)を設定する
`get_state()`	音声が再生中かどうかを知る

vlc の主な操作

VLC そのものの後に、Python ライブラリをインストールします。

```
$ sudo apt-get install vlc
:
:
$ sudo pip3 install python-vlc
```

Python 対話モードで動作を確認してみましょう。音声ファイルのあるディレクトリで実行します。

```
$ python
Python 3.7.3 (default, Dec 20 2019, 18:57:59)
>>> import vlc
>>> p =vlc.MediaPlayer()
>>> p.set_mrl('hello.wav')
<vlc.Media object at 0xb678e9f0>
>>> p.play()
0
>>> exit()
```

「こんにちは」と聞こえたら OK です。

発話の音量を上げたり下げたりする命令を追加することにします。ついでに、歌を一曲唄えるようにしましょう。

代表的な語りかけ	返事
もっと大きな声で話して	このくらいの声でどうですか
ちょっと小さな声にして	このくらいの声でどうですか
なにか歌って	「いつも何度でも」を歌います

追加する命令

python-vlc を使う上で問題がありました。root 権限で使えない（一部は動いて、一部は動かない！）動作があるのです。他のサブプロセスは GPIO や通信を使うので、root 権限（sudo）で起動させる必要があります。また、電源投入後に自動で立ち上げようとすると、root 権限になってしまいます。ネット上には VLC の実行ファイルに手を入れてごまかす方法が書かれていますが、正攻法を取り、voice.py だけは一般ユーザとして動かすことにします。インクルードファイル kaonashi.h を参照してください。

8.4.7時刻管理

指定時刻になったら教える機能で、タイマーとよく似ています。残念なことに Linux には「目覚まし時計」機能がないので、タイマープロセスを改造した実装が必要です。実装方法は次の二通りが考えられますが、タイマープロセスがそのまま使える選択肢 1 を選びました。

1. 起動時刻から指定時刻までの時間を計算し、その間スリープする
2. 適当な時間のスリープを繰り返し、指定時刻を過ぎたか確認する

アラーム起動用の命令を追加します。

代表的な語りかけ	返事
○分たったら知らせて（実装済）	○番タイマーを起動しました ○分後にお知らせします
○時○分になったら教えて	○番タイマーを起動しました ○時○分にお知らせします
（通知時間になったら（実装済））	○番タイマーの時間が来ました

時間管理の命令

8.4.8検索クラウドサービス

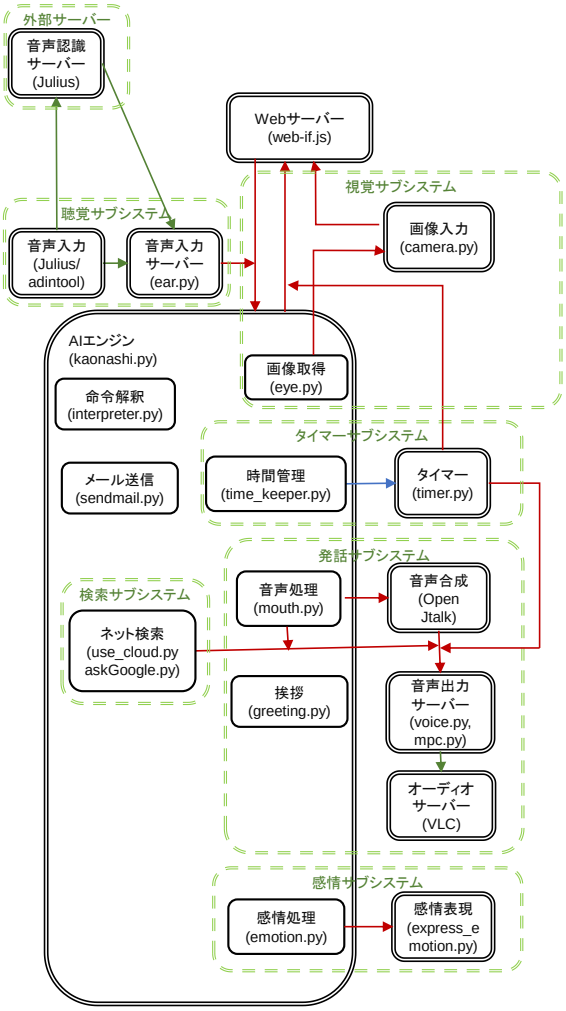
日本語テキストで要求を受け付けてくれることが分かっている、Google Assistant を使うことにします。Google Assistant は返事のテキストも返してくれるので、それを Web にも表示することを目論みました。

9　実証モデルの開発

前章の検討に基づいて実証モデル（フェーズ 4）を開発します。この章では、サブシステムやモジュールの設計情報をまとめますが、紙面の制約から、ソフトウエアモジュールの掲載や、詳しい検証の説明は、一部を除いて省略します。主要な部分は前章に出てきているし、検証には前章までの方法論がそのまま適用できるのからです。巻末の付録にあるプロジェクトファイルをダウンロードすれば、すべてのファイルを見ることができます。

9.1　システム構成

実証モデルのシステム構成（とモジュール）を次に示します。プロトタイプと比べて複雑になっていますが、ここまでの説明を読んでいれば、迷うことはないと思います。



実証モデルのシステム構成

9.1.1マルチプロセッサシステム設計上の注意点

音声認識 Julius を外部サーバーに常駐させると、秘書ロボットの機能は、二つのハードウエア上に分散することになります。一種のマルチプロセッサシステムで、各々が相手と協調して動作することが求められます。

秘書ロボットの場合は、Julius がサーバーになり、ear.py と adintool がクライアントとして処理や結果を要求します。うまく動いているときは良いのですが、相手方が期待したとおりの動作をしてくれないと、おかしなことが起こります。ロボット本体の起動やシャットダウンがあっても、Julius サーバーは常駐して接続を待つようにしたいので、注意が必要です。ロボットが勝手にシャットダウンすると、サーバーが接続しっぱなしになり、再度接続を試みると拒否することがありました。OS レベルで監視していないか、監視していても切断までの時間が長すぎるせいです。お行儀よく通信を切ってからシャットダウンした方が良さそうです。

実は、マルチプロセッサシステムでなくても、同じような問題は起こり得ます。デバイスドライバやライブラリを使っているとき、勝手にプロセスを終了すると、（主に通信）資源を解放しないため、以後はライブラリなどが使えなくなることがあります。今回はカメラのライブラリでそういう現象がおこりました。終了前に close して資源を解放するようにしました。

9.1.2プロセス ID を得るために

サブプロセスを起動すると、プロセス ID などが返ってくるので、これを使って停止命令を送るようにします。そのため、ここまでのやり方を変更します。

プロトタイプでは、確実に最新のプログラムコードを使うため、実行時に cpp で処理してから Python に渡していました。そのためのシェルスクリプトを呼ぶと、返ってくるのはシェルのプロセス ID で、さらにその子プロセスである Python のプロセス ID は分かりません。

そこで処理を二段階にし、Python が実行できるコードを生成してから、それを実行するように手順を作りました。一種のコンパイルなので、事前にやっておけば実行するだけで済むのですが、改造したとき

に処理を忘れてしまうのを懸念して、こういう手順をとりました。

事前に処理しておくのは、タイマープロセス（毎回だと煩雑だから）と HTML 文書だけにしています。

9.1.3子プロセスへの指令

プロセス ID の分かっている子プロセスに「通信を切れ」という指令を与えるのに、OS が提供している「シグナル」を使います。これは一種のソフトウエア割り込みで、子プロセス側に割り込み処理を記述しておけば、親プロセスは send_signal()を実行するだけです。ユーザが自由に使っていいシグナルが二つあるので、後ろの方（SIGUSR2）を使うことにしました。シグナルを使うためのインクルードファイル use-signal.h を用意しました。

親プロセスの方では、子プロセスの起動命令 fork（実際には subprocess.Popen）実行時の戻り値を記憶しておき、自分が停止する前に子プロセスに停止シグナル（自作プログラムには SIGUSR2、既存プログラムには終了シグナル SIGTERM）を送ります。

親プロセス（追加した子プロセスの処理部のみ抜粋）

```
#include “include/use-signal.h”
（中略）
/* 子プロセスの起動命令 */
child = fork(['python', '<子プロセス実行形>'])
（中略）
/* 子プロセスへの停止シグナル */
child.send_signal(SIGUSR2)
（後略）
```

子プロセスでは、シグナル受信時のサービス関数（右上の例では sig_hdlr）内に必要な処理を記述し、最後に終了（exit）するようにします。関数の引数は、シグナル番号とスタックフレーム（ここでは使わない）です。そのあとで、シグナル受信時の処理を、signal(シグナル番号, サービス関数名)で指定します。

親プロセスをキーボード割り込み（Ctrl+C）で終了させようとすると、子プロセスにも同じシグナル（SIGINT）が届き、放っておくと終了させられてしまいます。そのため SIGINT でも同じサービス関数が実行できるように指定する必要があります。その結果、子プロセス内で except KeyboardInterrupt を指定する必要はなくなります。

子プロセス（追加したシグナル処理部のみ抜粋）

```
#include “include/use-signal.h”
（中略）
def sig_hdlr(signo, frame): /* 終了シグナル処理 */
  /* 終了前に行う処理をここに記述する */
  sys.exit(0)

signal(SIGUSR2, sig_hdlr)   /* 終了要求シグナル */
signal(SIGINT, sig_hdlr)    /* Ctrl+C 割込 */
（後略）
```

実際のコードでは、使いやすいマクロを定義しているので、インクルードファイル kaonashi.h を参照してください。

9.1.4個人情報の一括管理

プロトタイプでは、あまり気にしてこなかったのですが、動作環境の定義（IP アドレスやディレクトリ情報など）が複数のファイルに分散していました。実証モデルでは、これに加えて個人情報（各種アカウント名やパスワード、サーバー名など）を使用します。皆さんが利用するのに便利なように、こういった情報をまとめて、インクルードファイル personal_info.h に記載するように改造しました。公開ファイルでは、一部の情報を削除したり変更したりしています。皆さんがプログラムコードを利用するときは、ご自分の情報を記入したり、動作環境に合わせて修正したりしてください。

9.2　聴覚サブシステム

聴覚サブシステムでは、「Julius のリモートサーバーがあれば、それを使う」ように改造しました。サーバーは複数用意し、望ましい（処理が速い）ものから使っていくようにしました。リモートサーバーが見つからなかったら、自前の Julius を使用します。

9.2.1インクルードファイル

tcp.h では、三台（設定を変えれば何台でも使えます）のリモートサーバーを定義し、その接続先（自身と adintool それぞれの）をリスト化しています。プロジェクトファイルを見てください。

9.2.2音声入力サーバー

音声入力サーバーは、複数の Julius リモートサーバーに対し、リストにある順に接続を試みます。接続に成功したサーバーのアドレスを adintool に教えてやります。すべてのリモートサーバーに接続できないときには、adintool ではなく、自前の Julius を起動するようにしています。

マルチプロセッサシステムの注意で述べた処理を実装します。親プロセスから OS を介して終了指示シグナル（SIGUSR2）を受信したら、adintool を終了させ、Julius との TCP 通信を切断してから、自分自身の処理を終えるようにしました。ここまでを別モジュール julius.py として ear.py から独立させました。

julius.py

```
  /* 聴覚サブシステム：音声認識部
     初版：2020/3/5 Chuji
    改定版：2020/4/27 --- Julius リモートサーバー追加
    最新版：2020/6/23 --- Julius 接続部を分離

  Class:
    julius_server
  属性：
    julius   ：Julius サーバーへ接続した TCP ソケット
    adc      ：マイク入力サブプロセス
    jserver  ：接続した Julius サーバーの名前
    Jservers  ：接続を試みる Julius サーバーのアドレスリスト
    Jnames  ：接続を試みる Julius サーバーの名前リスト
    Aservers  ：adintool に与える Julius サーバーの IP アドレス
  操作：
    connect  ：使用可能なサーバーと接続する
    terminate  ：サーバーとの接続を閉じる
    find_Julius  ：使用可能な外部 Julius サーバーを探す
  */

  #include "include/kaonashi.h"
  #include "include/tcp.h"
  #include "include/use-sys.h"
  #include "include/use-time.h"
  #include "include/use-signal.h"
  #include "include/julius.h"

  import socket       /* Julius との通信ポートを開く */

  class julius_server:
    def __init__(self):
      /* 使用可能な Julius サーバーのリスト */
      self.Jservers = JULIUS_SERVERS
      self.Aservers = ADINNETS
      self.Jnames = JULIUS_NAMES
      self.adc = None
      self.julius = None
      self.jserver = LOCAL_SERVER

    /* svr 番目の Julius サーバーが使えるか確認する */
    def find_Julius(self, svr):
      s = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
      s.settimeout(TCP_TIMEOUT) /* 存在を確認するためのタイムアウト */
      try:
        /* サーバーへの接続を試みる */
        s.connect(self.Jservers[svr])
        /* 接続に失敗すると異常処理に飛び、以下は実行されない */
        s.settimeout(None) /* 接続成功後はタイムアウトなし */
  #ifdef DEBUG
        print('Julius on ', self.Jnames[svr], 'is
available')
  #endif
        return s    /*接続に成功した場合はソケットを返す */

      except socket.timeout: /* 接続できなかった場合 */
  #ifdef DEBUG
        print('Server on ', self.Jnames[svr], 'is
not available')
  #endif
        s.close()
        return None

      except socket.error: /* 接続が拒絶された場合 */
  #ifdef DEBUG
        print('Julius on ', self.Jnames[svr],
'refused connection')
  #endif
        s.close()
        return None

    /* 優先度に従って Julius サーバー（外部・内部）に接続する */
    def connect(self):
      for server in range(NUM_JULIUS):
        self.julius = self.find_Julius(server)
        if self.julius != None: /*接続成功 */
          self.jserver = self.Jnames[server]
          /* 音声入力を起動する */
  #ifndef MANUAL_ADIN
          self.adc =
fork(RUN_ADINNET(self.Aservers[server]))
  #endif
          break
      /* すべてのリモートサーバーに接続できなかったら */
      if self.jserver == LOCAL_SERVER:
        /* 自分の中でサーバーを動かす */
        self.adc = fork(RUN_MY_JULIUS)
        sleep(WAIT_FOR_JULIUS_TO_READY)
        self.julius = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
        self.julius.connect(JULIUS_LOCAL)

  #ifdef DEBUG
      print('Julius in on ', self.jserver)
  #endif
      return self.julius

          /* 接続終了処理 */
    def terminate(self, signo, frame):
  #ifndef MANUAL_ADIN
      self.adc.send_signal(SIGTERM) /* マイク入力を切る */
  #endif
      if self.jserver != LOCAL_SERVER: /* リモート Julius 接続時は */
        self.julius.close() /* クライアントから接続を切る */
      sys.exit(0)
```

ear.py（変更箇所のみ赤字で示す）

```
  /* 聴覚サブシステム：音声入力サーバー
     初版：2020/3/5 Chuji
    最新版：2020/4/27 --- Julius リモートサーバー追加
  */

  #include "include/kaonashi.h"
  #include "include/tcp.h"
  #include "include/use-sys.h"
  #include "include/use-time.h"
  #include "include/use-signal.h"
  #include "include/julius.h"

  #ifndef HEARING_TEST /* HEARING TEST では、聞取り結果を AI に渡さない */
  #include "include/use-redis.h"
  fifo = open_FIFO()
  #endif

  #include "julius.py" /* Julius サーバー接続処理 */
  #include "julius-xml.py" /* XML 文書の解析部 */

  jserver = julius_server()
  julius = jserver.connect()

  signal(SIGUSR2, jserver.terminate) /* シグナルハンドラーの定義 */
  signal(SIGINT, jserver.terminate)

  xml = xml_interpreter()
   （後略）
```

ear.py の前半で Julius サーバーへ接続したら、後半はプロトタイプと同じです。検証条件を#define して、動作を確認します。

```
$ cpp -DDEBUG -DHEARING_TEST -DTEST_LINE ear.py |
sudo python
```

実証モデルでは、Julius の出力を解釈する xml.py の名前を julius-xml.py に改名しました。XML 文書を扱う他のライブラリ（GoogleAssistant SDK が使っている）との混同を防ぐためです。

Julius の処理が早くなって、julius-xml.py のバグが見つかりました。検出した日本語が、次の処理終了で上書きされて、消えてしまうことが分かりました。検出した日本語をリストに入れることで回避しました。また、テキストの解析に正規表現ライブラリを利用することで、記述しやすくしました。

julius-xml.py（変更箇所のみ赤字で示す）

```
  （略）

  #include "include/julius.h"

  import re  /* 正規表現ライブラリを使う */

  class xml_interpreter:
    def __init__(self):
      self.xml = XML_EMPTY  /* 受信した XML 文書（一部）
*/
      self.text = XML_EMPTY  /* 認識した日本語の一部 */
      self.japanese = XML_EMPTY_LIST  /* 認識した日本
語全文 */

  （中略）
    /* XML 文書から一行づつ解析し、認識文字列があったら連結
していく
       先頭がピリオドの行があったら、解析を終了 */
    def parse(self):
      line = self.getline()
      while line != XML_EMPTY:
  #ifdef TEST_LINE
        print (line)
  #endif
        if line[XML_FIRST_CHAR] == XML_DELIMITER:
          if self.text:
            self.japanese.append(self.text)
            self.text = XML_EMPTY
        elif XML_TEXT in line:
          fragment = Get_Japanese(line)
          if fragment == XML_END_VOICE:
              self.text += XML_PERIOD
          elif fragment != XML_START_VOICE:
              self.text += fragment
        line = self.getline()
      return

    /* 解析が終了した日本語文を返す。ないときは空文字列を返す
       日本語文を返したら、次の解析を始める */
    def get_japanese(self):
      if self.japanese:
        return self.japanese.pop(0)
      else:
        return XML_EMPTY
```

9.3　発話サブシステム

プロトタイプでは、「音声出力クライアント」という呼び方をしていました。実際に音声出力を行うソフトウエアを「サーバー」と呼ぶため、その「クライアント」という位置づけだったからです。しかし、秘書ロボットのサブシステムとしての位置づけは、「音声出力を行うサーバー」なので、実証モデルからはサーバーと呼びなおすことにします。音声合成 resident_open_jtalk はプロトタイプをそのまま使用します。

9.3.1音声処理モジュール mouth.py

AI エンジンの一部である音声処理モジュール mouth.py は、プロトタイプに簡単な修正を加えます。まず、再生音量の変更ができるように、操作を追加しました。

定型文の辞書は、定義を AI_commmands.h に移して、それを読み取るように変更しました。辞書を増補するとき、AI_commmands.h の修正で済ませるためです。

クラウドサービス音声の処理 say_this を用意していましたが、発話タイミングを確保する WAIT 発行操作 reserve を追加して使います。また、音楽などの再生のみを行い、Web へはテキストを表示しない play_this も追加しました。

検証は test_mouth.py に追加しました。辞書追加分の検証は省略しています

9.3.2音声出力サーバー

voice.py

ステートマシン voice.py には音量調節機能を追加しました。要求を mpc.py に渡すだけなので大きな修正ではありません。

voice.py の検証は、mpc.py の検証が済んだ後に、音量調整機能の検証を追加した test_voice.txt の内容を Redis キーREDIS_TO_MPC に送ることで行います。

mpc.py

このモジュールは、プログラムの構造と機能を保ったまま、実際の処理を置き換えます。aplay を起動する代わりに、VLC に要求を送りこむように変えていきます。このとき、各々の音声ファイルの再生が終わるのを待ってから、次のファイルを再生することで、順番に再生するようにタイミングを調整して

います。直接 VLC を操作しているので、マクロ定義 SIMULATION は使いません。

mpc.h（変更箇所のみ赤字で示す）

```
  （略）

  /* pygame.mixer 用命令 */
  #define SOUND_PLAYING  3 /* VLC 演奏中ステータス */
  #define SOUND_ENDED  6 /* VLC 演奏終了ステータス */
  #define SOUND_PACKAGE  vlc
  #define SOUND_LIB  v
  #define SOUND_INIT  self.s = v.MediaPlayer()
  #define SOUND_ADD(x)  self.s.set_mrl(x)
  #define SOUND_PLAY  self.s.play()
  #define SOUND_BUSY  self.s.get_state() !=
SOUND_ENDED
  #define SOUND_MAX  100 /* 最大音量 */
  #define SOUND_MIN  0 /* 最小音量 */
  #define SOUND_INC  20 /* 音量調整単位 */
  #define SOUND_GET_VOL  self.s.audio_get_volume()
  #define SOUND_UP_VOL(x)
self.s.audio_set_volume(min(x + SOUND_INC,
SOUND_MAX))
  #define SOUND_DN_VOL(x)
self.s.audio_set_volume(max(x - SOUND_INC,
SOUND_MIN))

  #define SOUND_INTERVAL  0.25 /* 再生中を確認する間隔
*/
  （後略）
```

mpc.py

```
  /* 音声再生サーバー（旧 mpc クライアント）
    初版：2020/3/10 Chuji
    最新版：2020/5/14 --- VLC による再生に全面書き換え

  クラス：mpd_client

  属性：
    SOUND_LIB:  python audio ライブラリ
    queue:  再生ファイルの待ち行列

  操作：
    add:  プレイリストに音声ファイルを追加ｓる
    play:  プレイリストを再生する
    volume_up:  再生音量を上げる
    volume_down:  再生音量を下げる
  */

  #include "include/AI_commands.h"
  #include "include/mpc.h"
  #include "include/use-time.h"
  #include "include/use-sys.h"

  import SOUND_PACKAGE as SOUND_LIB

  class mpd_client:
    def __init__(self):
      SOUND_INIT
      self.queue = MPC_Q_EMPTY

    /* 音声ファイルをリストに加える */
    def add(self, sound):
      self.queue.append(sound)

    /* リストにある音声ファイルを順番に再生する */
    def play(self):
      while (len(self.queue) > 0):
        sound = VOICE_DIRECTORY +
self.queue.pop(MPC_Q_TOP)
        if os.path.exists(sound):
          SOUND_ADD(sound)
          SOUND_PLAY
          sleep(SOUND_INTERVAL)
          while SOUND_BUSY:
            sleep(SOUND_INTERVAL)
```

```
    def volume_up(self):
      v = SOUND_GET_VOL
  #ifdef DEBUG
      print('volume = ', v)
  #endif
      SOUND_UP_VOL(v)

    def volume_down(self):
      v = SOUND_GET_VOL
  #ifdef DEBUG
      print('volume = ', v)
  #endif
      SOUND_DN_VOL(v)
```

検証は、以前と同じ test_mpc.py で行います。音量調整の検証を付け加えました。マクロ定義 SIMULATION を使わないので、音声出力で確認します。

VLC 版の前に、pygame を使ったモジュールを設計していました。サンプリングレートの問題が分かってから VLC に乗り換えたのですが、実は mpc.py は（コメントを除いて）全く書き換える必要がなく、インクルードファイル mpc.h の定義を変えるだけで済みました（「pygame 用」というコメントが、まだ残っている）。プログラムの構造は変わっていないので、デバッグは実に簡単でした。

9.4　視覚サブシステム

視覚サブシステムは前章で試験に使った test_camera.py を改造して使います。

9.4.1インクルードファイル

インクルードファイル vision.h では、カメラの設定、ファイル名、コマンドなどを定義しています。また画像入力プロセスにコマンドを送る Redis キー REDIS_TO_CAMERA を redis.h に追加しました。

vision.h

```
  /* カメラ操作定義
    初版：2020/4/29 Chuji
    最新版：
  */

  #ifndef __CAMERA

  /* 画素数 */
  #define NORMAL_RESOLUTION  (640, 480)
  #define MEDIUM_RESOLUTION  (1280, 960)
  #define FILE_RESOLUTION  (2560, 1920)

  /* ファイル名 */
  #define SHOT_FILES  5
  #define SNAP_SHOT  'image'
  #define IMAGE_EXT  '.jpg'
  #define STILL_FILE  'still.jpg'

  /* 待ち時間 */
  #define SNAP_INTERVAL  0.6
  #define PHOTO_INTERVAL  5

  #define SHUTTER_TIMING  3  /* 「はいチーズ」と言うまで
の時間 */
  #define SHUTTER_TIMING2  2  /* シャッター音が出るまで
*/
```

```
/* カメラ駆動コマンド */
#define START_MOVIE  'Start'
#define STOP_MOVIE  b'Stop'
#define STILL_PHOTO  'Still'
#define HD_PHOTO  'HD'

/* 画像入力プロセスの起動 */
#define CAMERA_ON  './run_camera'

#define __CAMERA

#endif
```

9.4.2画像取得モジュール

画像取得モジュール eye.py は、AI エンジンの一部として、カメラを駆動している画像入力プロセスに命令を送る役目を果たします。そのほか、Web や音声で返事をしたり、撮影タイミングを示す LED の点滅を指示したりします。

eye.py

```
/* eye.py  画像取得モジュール
   初版： 2020/4/29/ Chuji
   最新版：


Class: Photographer
属性:
  fifo:  Redis FIFO
  lip:  音声処理オブジェクト
  led:  LED 表示（感情処理）オブジェクト
操作:
  movie_start:  動画撮影を始める
  movie_stop:  動画撮影を終了する
  still_photo:  静止画を撮影する
  HD_photo:  高解像度の静止画を撮影する
*/

#include "include/use-redis.h"
#include "include/use-time.h"
#include "include/vision.h"


/* 画像取得オブジェクト */

class Photographer:
  def __init__(self, mouth, led):
    self.fifo = open_FIFO()
    self.lip = mouth
    self.led = led
  /* 各操作は、発話・発話内容の Web 表示、カメラの操作からなる */
  /* 動画の撮影・配信開始 */
  def movie_start(self):
    self.lip.ah()
    self.lip.say(WORD_MOVIE_START)
    self.lip.ahah()
    self.fifo.rpush(REDIS_TO_CAMERA, START_MOVIE)

  /* 動画の撮影・配信停止 */
  def movie_stop(self):
    self.fifo.rpush(REDIS_TO_CAMERA, STOP_MOVIE)
    self.lip.ah()
    self.lip.say(WORD_MOVIE_STOP)
    self.lip.ahah()

  /* 写真撮影の案内とタイミング制御 */
  def shutter_timing(self):
    self.lip.ah()
    self.lip.say_now(WORD_TAKE_PHOTO)
    sleep(SHUTTER_TIMING)
    self.led.camera()
    sleep(SHUTTER_TIMING)
    self.lip.say(WORD_CHEESE)
    self.lip.click()
    sleep(SHUTTER_TIMING2)

  /* 写真撮影後の案内 */
  def after_shutter(self):
    sleep(SHUTTER_TIMING)
    self.lip.say(WORD_PHOTO_TAKEN)
    self.lip.ahah()

  /* 写真を撮影する */
  def still_photo(self):
    self.shutter_timing()
    self.fifo.rpush(REDIS_TO_CAMERA, STILL_PHOTO)
    self.after_shutter()

  /* 高解像度写真を撮影する */
  def HD_photo(self):
    self.shutter_timing()
    self.fifo.rpush(REDIS_TO_CAMERA, HD_PHOTO)
    self.after_shutter()
```

静止画撮影では、写る人が準備をする時間を作るため、案内の発語と LED の点滅を制御してから撮影します（シャッター音を出します）。画像ファイル名はひとつだけで、命令があったら他のモジュールがメールに添付します。

検証は、画像入力プロセスの検証が済んでから、test_eye.py を使い、機能をすべて確認します。

9.4.3画像入力プロセス

画像入力プロセス camera.py は、カメラを駆動して写真を撮影し、Web に表示します。動作モードは以下の三種類です。

- 中解像度での連続撮影
- 中解像度での一回撮影
- 高解像度での一回撮影

連続撮影では、画像ファイル名を使いまわしながら表示を繰り返します。停止命令はいつ来るか分からないので、撮影の前に Redis から取得（ポーリング）しようとします。このとき、命令が来ていなくても動作が止まらないように、blpop ではなく lpop を使っています。

静止画はすぐに撮影します。

camera.py

```
/* カメラによる撮影制御
     初版：2020/4/16 Chuji
   最新版：
*/

#include "include/use-sys.h"
#include "include/use-signal.h"
#include "include/use-redis.h"
#include "include/use-time.h"
#include "include/vision.h"

from picamera import PiCamera

camera = PiCamera()
camera.resolution = NORMAL_RESOLUTION

def terminate(signo, frame):
```

```
    camera.close()
    sys.exit(0)

  signal(SIGUSR2, terminate)
  signal(SIGTERM, terminate)


  /* 画像ファイル名を使いまわすためのカウンター */
  image = 1

  fifo = open_FIFO()  /* Redis FIFO で結果を知らせる準
備 */

  /* 写真を撮影して、画像ファイル名を Web サーバーに知らせる
*/
  def capture(file_name):
    camera.capture(file_name)
    fifo.rpush(REDIS_TO_DISP, file_name)

  /* 処理本体 */
  try:
    while True:
      /* FIFO から指令を読み取る */
      tag, req = fifo.blpop(REDIS_TO_CAMERA)
      req = req.decode()
  #ifdef DEBUG
      print('command = ', req)
  #endif
      /* 写真の撮影指令 */
      if req == STILL_PHOTO:
        capture(STILL_FILE)

      /* 高解像写真の撮影指令 */
      elif req == HD_PHOTO:
        camera.resolution = MEDIUM_RESOLUTION
        capture(STILL_FILE)
        camera.resolution = NORMAL_RESOLUTION

      /* 動画（連続写真）の撮影開始指令 */
      elif req == START_MOVIE:
        while True:
          /* ファイル名を使いまわして撮影 */
          file_name = SNAP_SHOT + str(image) +
IMAGE_EXT
          capture(file_name)
          image += 1
          if image > SHOT_FILES:
            image = 1

          /* 適当な間隔を置いて FIFO に停止指令が来ていない
か調べる */
          sleep(SNAP_INTERVAL)
          req = fifo.lpop(REDIS_TO_CAMERA) /* non-
blocking */
  #ifdef DEBUG
          print('popped =', req)
  #endif
          if req == STOP_MOVIE:
            break
```

検証では pusher.py を使って、vision.h で定義したコマンドを送り、Web 上で画像が更新されることを確認します。

9.5　メール送信サブシステム

メール送信サブシステムは AI エンジンの処理モジュールとして設計しますが、画像入力と関係があるので、ここで説明します。

メールの受信には、受信メール一覧の表示と選択、不要メールの削除、添付ファイルの処理、スパム対策など、複雑な処理があります。これをすべて音声インターフェースだけで実装するのは面倒です。いっぽう送信は定型的な処理が多いので、安心してロボットに任せられます。スパムを発信しないよう気をつける必要はありますが。

9.5.1インクルードファイル

メール送信に必要な定義を smtp.h に記述しておきます。個人情報であるアカウント名、パスワードや送信サーバーを記述するので、漏洩しないよう気をつけてください。公開したファイルでは、そのような箇所に手を入れています。

SMTP（Simple Mail Transfer Protocol）は、メールサーバーへ電子メールを送信するプロトコル（通信手順）のことです。

smtp.h

```
  /* 電子メール関連の定義
     初版：2020/4/29 Chuji
     最新版：2020/8/8 --- 個人情報を移動
  */

  #ifndef __SMTP

  #include "personal_info.h"

  /* サーバー情報 */

  #define SMTP_SERVER  MY_SMTP_SERVER /* 各自の SMTP
サーバー */
  #define SMTP_PORT  587 /* SMTP ポート番号 */

  /* アカウント情報 */

  #define EMAIL_ACCOUNT  MY_EMAIL_ACCOUNT
  #define EMAIL_PASSWORD  MY_EMAIL_PASSWORD

  /* MIME 要素 */

  /* フィールド名 */
  #define SENDER_FIELD  'From'
  #define RECEIVER_FIELD  'To'
  #define SUBJECT_FIELD  'Subject'
  #define CC_FIELD  'Cc'
  #define BCC_FIELD  'Bcc'
  #define DATE_FIELD  'Date'
  #define EMPTY_FIELD  ''

  /* ファイル要素の記述内容 */
  #define MIME_FILE  'rb'
  #define UNIX_CHAR  'utf-8'
  #define TEXT_MIME  'plain'

  /* 定型メール定義 */

  #define MAIL_RECEIVER  MY_EMAIL_RECEIVER
  #define MAIL_TITLE  'カオナシが撮影した写真'
  #define MAIL_BODY  'さっき撮影した写真をお送りします。'

  #define __SMTP
  #endif
```

9.5.2メール送信モジュール

メール送信モジュール sendmail.py は、画像ファイルを送信するだけなので、オブジェクトではなく、関数として設計しました。送信に必要なライブラリは Raspbian に付属しているので、インストールの必要はありません。

もともとSMTPは英文を送る手順として開発されましたが、日本語やファイルも送れるように拡張されたMIME（Multipurpose Internet Mail Extensions）を使います。本文や添付など複数の部分を含めるよう、Multipartとしてオブジェクトを生成します。

最初に送り先などを追加し、次に本文を日本語テキストとして追加します。それから画像ファイルをエンコードした添付テキストを追加します。ファイルが画像であることを仮定しているので、いきなりMimeImageで処理しています。他のファイルも送れるようにするには、ファイルの型によって指定を変えるので、もう少し面倒な処理が必要です。

最後にSMTPサーバーにログインして、メールを送信します。サーバーのポート番号やセキュリティプロトコル（TLS: Transport Layer Security）が異なるサーバーの場合は、それに従います。

sendmail.py

```
  /* 電子メール送信モジュール
      初版：2020/4/29 Chuji
    最新版：

  関数: sendEmail
    send:  定型メールの送信

  */

  #include "include/smtp.h"

  from smtplib import SMTP
  from email.mime.multipart import MIMEMultipart
  from email.mime.text import MIMEText
  from email.mime.image import MIMEImage


  def sendEmail(attachment):
    /* マルチパートメールを生成する */
    msg = MIMEMultipart()

    /* 送付先等を設定する */
    msg[SUBJECT_FIELD] = MAIL_TITLE
    msg[SENDER_FIELD] = EMAIL_ACCOUNT
    msg[RECEIVER_FIELD] = MAIL_RECEIVER

    /* 定型本文を付加する */
    msg.attach(MIMEText(MAIL_BODY, TEXT_MIME,
UNIX_CHAR))

    /* 指定ファイルを添付する --- 画像ファイルのみサポート
*/
    with open(attachment, MIME_FILE) as fp:
      img = MIMEImage(fp.read())
    msg.attach(img)

    /* メールを送信する */
    server = SMTP(SMTP_SERVER, SMTP_PORT)
    server.starttls()  /* 暗号化プロトコルを起動 */
    server.login(EMAIL_ACCOUNT, EMAIL_PASSWORD)
    server.send_message(msg)
```

添付ファイルを指定してこの関数を呼べば、メールが送信されます。各自検証してみてください。

9.6　タイマーサブシステム

タイマーサブシステムには、時刻管理（目覚時計）を追加します。インクルードファイル（timer.h）、タイマー機能（timer.py）に変更はありません。

9.6.1時間管理モジュール

AIエンジンの時間管理モジュールtime_keeper.pyに、新しい操作set_alarmを追加して、目覚ましをかけます。午後に12時間制の時刻を設定したら、24時間制に直して時間差を計算します。

9.6.2時間取得モジュール

時間取得モジュールget_time.pyにも処理を追加しました。いずれも時刻の表現に特有の処理です。

- 午後3時という表現を見つけたら、24時間制の15時を返す
- 3時5分前という表現を見つけたら、2時55分を返す

時間を「分」で返す操作get_time()に加え、「時（間）」と「分」に分けて結果を返す操作get_clock()を追加しました。

更新した評価用スタブtest_gettime.pyと、テストパターンを記述したテキストファイルtest_gettime.txtはプロジェクトファイルに含まれています。

9.7　Web サーバー

Webサーバーweb-if.jsとHTML文書index.html.sourceは、前章で会話履歴と画像表示を追加したものを、ほぼそのまま使います。

変更したのは、会話履歴領域の位置と大きさだけです。Google Assistantの返事（それまでの例より長い時がある）も表示したいので、領域からはみ出すことも考慮しました。

9.8　感情サブシステム

感情サブシステムは、AIエンジンの感情処理モジュール（emotion.py）と常駐プロセス（express_emotion.py）からなります。

9.8.1インクルードファイル

インクルードファイルemotion.h（プロジェクトファイルを見てください）は、冒頭にあるカラーLEDモジュールの定義（最大輝度による制限はしていません）以外、全面的な改造になりました。

前半は感情処理モジュールが使う「気分変数」を決定するパラメータと変化幅を定義しています。気分変数の値が 7 つの領域（BEST, BETTER, GOOD, NORMAL, BAD, WORSE, WORST）のどこにあるかにより、異なった感情表現を求めます。

そのあとに続く色データは、感情表現に使うもので、感情ごとにデザインした発光シナリオの色を定義しています。

感情表現を常駐プロセスにしたので、Redis FIFO のキーを redis.h に追加しました。

9.8.2感情処理モジュール

AI エンジン感情処理モジュール emotion.py（プロジェクトファイルを参照してください）は初期化（感情表現プロセスの起動を含みます）のあと、AI エンジンの中から、6 種類の操作（Best, Better, Good, Bad, Worse, Worst）で呼び出されます。それぞれに対応する変化幅だけ、処理操作 show で気分変数を変化させます。

その時、前回の気分が最高（BEST）か最低（WORST）だったら、NORMAL の GOOD 側あるいは BAD 側境界まで、時間に応じて減衰していくようになっています。つまり、最高の気分も最低の気分も、時間が経てば醒めてくるという現象を模擬しています。

最後に、新しい感情変数値が 7 つの領域のどこにあるかによって、別々の感情表現をするように指令します。

9.8.3感情表現プロセス

感情表現プロセス express_emotion.py プロジェクトファイルを参照してください）は、Redis FIFO（REDIS_TO_EMOTION）を介して、感情処理モジュールから 7 種類の感情を受け取り、対応する発光シナリオ（happy, great, good, calm, bad, despair, mad）を実行します。

カメラ撮影用に、セルフタイマーを模擬するシナリオも用意しました。赤い点滅を 3 回繰り返します。

各シナリオは、発光パターンと間隔を変えることで実現しています。詳細な説明は省略します。

9.9　検索サブシステム

検索サブシステムは、AI エンジンの処理モジュールの一つとして実装しました。実際には、

- Google Assistant を呼び出す検索実行モジュール askGoogle.py と、
- 応答の手順を記述する検索応答モジュール use_cloud.py

に分けてあります。前者は Google Assistant のことだけ知っていれば分かるようにしました。Google Assistant にテキストの要求を渡すと、返事が音声とテキストで返ってきます。

後者はカオナシ固有の応答方法だけを知っていれば良いようにしました。テキストと音声を、それぞれ Web サーバーと音声出力サーバーに渡すのが役割です。このモデルでは、事前に用意しておいたテキストを送りつけます。

9.9.1ライブラリのインストール

まず、Google Assistant SDK のライブラリをインストールします。開発か趣味用なら、一時間あるいは一日当たりの回数制限内であれば、無料で使えます。

最初に汎用開発ツールを、次に SDK と認証ツールを以下の手順（英語）に従ってインストールします（https://developers.google.com/assistant/sdk/guides/service/python/embed/install-sample）。仮想環境を使うことが推奨されていますが、この Raspberry Pi ZERO はカオナシ以外に使わないので、直接インストールしました。

```
$ apt-get update
$ sudo pip3 install setuptools wheel
$ sudo apt-get install portaudio19-dev libffi-dev
libssl-dev

$ sudo pip3 install --upgrade google-assistant-
sdk[samples]
$ pip install --upgrade google-auth-oauthlib[tool]
```

google-assistant-sdk のインストールでは、C++で記述したプログラムを実行して、Python プログラムを生成するので、非常に長い時間（Raspberry Pi ZERO では一時間以上）がかかります。暴走したという心配をせずに、忍耐強く待ってください。

9.9.2JSON

唐突ですが、データ交換に使う JSON について、ここで簡単に説明しておきます。JavaScript Object Notation といって、データの交換に使う、JavaScript に似た書式のテキストです。機械が処理でき、人も（なんとか）読める形式ということで、よく使われるようになりました。

大雑把に言うと、{名前：値, 名前：値 ……} という並びになっており、値の方は、単純な値だったり、配列だったり、JSON だったりします。以下に例を挙げます。C 言語の経験のある人なら、値部のサイズが可変な「構造体」と言えば分るでしょうか。

JSON の例（上と下は同じものを表している）

```
{
   “hight”:175,
   “weight”:65,
   “name”:{
      “surname”:”Yamada”,
     “lastname”:”Taro”
   }
 }
```

```
{“hight”:175,“weight”:65,“name”:{“surname”:”Yamada”,“lastname”:”Taro”}}
```

Python でよく似たデータ構造は「辞書」です。Python プログラムで辞書として処理することがあります。このとき、テキストである JSON との間で、以下のような変換が必要になります。

JSON テキスト（Python のテキスト型変数）

json.loads ↓　↑ json.dumps

Python の辞書型変数

（読み込み）json.load ↑　↓ json.dump（保存）

JSON テキスト（ファイル）

9.9.3Google Assistant API

カオナシのような「組込み機器」のために用意された API（Application Program Interface:インターフェース外部仕様）は、下記の文書（英語）に説明されています。リクエスト・データを送りつければ、レスポンス・データを返してくれます。
https://developers.google.com/assistant/sdk/reference/rpc/google.assistant.embedded.v1alpha2#google.assistant.embedded.v1alpha2.EmbeddedAssistant

各々のデータ構造を「JSON もどき」にまとめてみました。ここで「もどき」というのは、以下の理由からです。

- 読みやすくするため、名前を引用符で囲まない
- ‘<’と’>’で囲んだ部分は値に関する説明文

検証段階で Google Assistant に送るデータを調べるとき、この表が役に立ちます。現在地の緯度と経度は省略しても良いのですが、天気予報とか「近くの○○」という検索に必要なので、付けることにします。

AssistRequest

```
{
  config:{
    audio_out_config{
      encoding:<以下のどれか[
        ENCODING_UNSPECIFIED,
        LINEAR16,
        MP3,
        OPUS_IN_OGG
      ]>,
      sample_rate_hertz: <32 ビット整数(16000-24000 Hz)>,
      volume_percentage: <32 ビット整数(1 - 100 in %)>
    },
    screen_out_config:{
      screen_mode:<以下のどれか[
        SCREEN_MODE_UNSPECIFIED,
        OFF,
        PLAYING
      ]>
    },
    dialog_state_in:{
      conversation_state: <直前のレスポンスで返されたバイト列>,
      language_code: <テキスト（日本語の場合は"ja-JP"）>,
      device_location:{
        coordinates:{
          lattitude: <倍精度浮動小数点数(緯度：-90.0～+90.0)>,
          longitude: <倍精度浮動小数点数 (経度：-180.0～+180.0)>
        }
      },
      (optional) is_new_conversation: <論理値>
    },
    device_config:{
      device_id: <登録時に与えられた機器 ID>,
      device_model_id: <登録時に与えられたモデル ID>
    },
    debug_config:{
      return_debug_info: <デバグ情報が必要かどうかの論理値>
    },
    <以下のどちらか一方(union) [
      audio_in_config:{
        encoding:{<以下のどれか[
          ENCODING_UNSPECIFIED,
          LINEAR16,
          FLAC,
        ],
        sample_rate_hertz: <32 ビット整数(16000-24000 Hz)>
      },
      text_query: <問い合わせるテキスト>
    ]>
  },
  audio_in: <バイト列（入力音声データ）>
}
```

同じく、Google Assistant が返すレスポンス・データの構造です。大事なのは返事の音声 audio_out と、テキスト supplemental_display_text です。

AssistResponse

```
{
  event_type:<以下のどれか[
    EVENT_TYPE_UNSPECIFIED,
    END_OF_UTTERANCE
  ]>,
  audio_out:{
    audio_data: <バイト列（出力音声データ）>
  },
  screen_out:{
    format:{<以下のどれか[
      FORMAT_UNSPECIFIED,
      HTML
    ]>,
    data: <バイト列（表示データ）>
  },
  device_action:{
    device_request_json: <JSON データ（機器側に要求する動作）>
  },
  speech_results:<以下の要素のリスト{
    transcript: <認識したテキスト>,
```

```
    stability" <浮動小数点数（認識の確度）>
  }>,
  dialog_state_out:{
    supplemental_display_text: <音声データのテキスト>,
    conversation_state: <バイト列（次のリクエストで使
う）>,
    microphone_mode:<以下のどれか[
      MICROPHONE_MODE_UNSPECIFIED,
      CLOSE_MICROPHONE,
      DIALOG_FOLLOW_ON
    ]>,
    volume_percentage: <32 ビット整数(1 - 100%)>
  },
  debug_info:{
    aog_agent_to_assistant_json: <JSON データ（デバグ
情報）>
  }
}
```

9.9.4Google への登録

Google Assistant のサービスを受けるには、「開発プロジェクト（プロジェクト ID」」と「開発する機器（モデル ID)」、それに「試作機（機器 ID)」を登録しておく必要があります。Google のアカウント（Gmail などで使う）が必要です。

登録は、プロジェクトマネージメントページ（https://console.actions.google.com/）から始めます。手順の説明は以下（英語）にありますが、インターフェースなどが頻繁に更新されるので、インターネット上のなるべく新しい記事を参考にしてください。大事な点だけを説明します。（https://developers.google.com/assistant/sdk/guides/service/python/embed/config-dev-project-and-account)

- 最初に「新しいプロジェクト」を登録します。プロジェクト名を（言語と所在地も）決めてやれば、「プロジェクト ID」を提案してきます（自分で書き換えることもできます)。この ID を控えておきます。
- 次に「プロジェクトの内容」を例から選べと求められますが、無視してください。ここで間違えると迷路に入ってしまいます。選択肢の下の方にある「機器の登録」を選んでください。
- 「製品名」や「メーカー名」には適当な名前を決めて入力します。「機器のタイプ」はどれを選んでも構いません。「モデル ID」を提案してくる（自分で書き換えられます）ので、この ID も控えておきます。
- Google Assistant API の有効化画面で、API を有効にします。
- 認証画面で client_secret_〇〇.json という長い名前のクライアント情報ファイルをダウンロードします。このファイル名を変更してはいけません。ここまでは PC 上で行えます。
- 次は機器（ロボット）の登録です。上のクライアント情報ファイルを Raspberry Pi ZERO に転送します（どこでも構いません)。Raspberry Pi ZERO 上（Tera Term など、画面のコピーができるクライアントで SSH 接続します）で、先にインストールした認証ツール google-oauthlib-tool を実行します。このとき先のクライアント情報ファイルの置き場所と名前を与えます。
- すると、「次の URL で、このアプリケーションを承認してください」という表示が現れます。この URL をコピーして、PC のブラウザに与えます。承認すると、認証コードを表示します。この認証コードをコピーして、SSH 窓に与えれば、ロボットの認証ファイル credentials.json を作って、保存先（~/.config//google-oauthlib-tool/の下）を教えてくれます。
- 次にサンプルプログラム googlesamples-assistant-pushtotalk にプロジェクト ID とモデル ID を与えて実行します。Enter キーを押してから、何か話しかければ、Google Assistant に接続します。このときの警告は無視しても良いようです。メッセージの中に、「この機器を次のインスタンス ID で登録しました」と表示されるので、この長い英数字テキストを「機器 ID」として控えておいてください。

以上で登録は終わりです。これ以降で使うのは、モデル ID、機器 ID、credentials.json だけです。

9.9.5インクルードファイル

インクルードファイルでは、ライブラリのインポート、お題目のように繰り返す長い名前の省略形、モデル ID などのアカウント情報、設定の固定値や初期値を定義します。このうちアカウント情報や自宅の緯度・経度は個人情報なので、プロジェクト公開ファイルや下のリストから割愛しています。personal_info.h にみなさんの情報を入れてください。

use-google.h

```
/* use-google.h -- interface to Google Assistant for
python
    初版:  2020/6/29
    最新版：2020/8/8 --- ユーザー情報を移動
*/

#ifndef __GOOGLE

#include "personal_info.h"
```

```
  /* Python ライブラリのインポート（注：Python コード） */

  #include "use-json.h"

  import google.auth.transport.grpc
  import google.auth.transport.requests
  import google.oauth2.credentials

  from google.assistant.embedded.v1alpha2 import (
      embedded_assistant_pb2,
      embedded_assistant_pb2_grpc
  )

  from google.type import latlng_pb2


  /* Google アカウント情報は個人情報ファイル personal_info.h
に移動 */

  #define MY_AUDIO  'GoogleReply.mp3'
  #define READ_FILE  'r'
  #define WRITE_FILE  'wb'

  /* Google Assistant API */
  #define GA_API  embedded_assistant_pb2
  #define GA_API_REFRESH()
google.auth.transport.requests.Request()
  #define GA_API_OPEN(x)
embedded_assistant_pb2_grpc.EmbeddedAssistantStub(x)
  #define GA_API_SECURE(x)
google.oauth2.credentials.Credentials(token=None,
**json.load(f))
  #define GA_API_CHANNEL(x, y, z)
google.auth.transport.grpc.secure_authorized_channel(x,
y, z)
  #define GA_API_ENDPOINT
'embeddedassistant.googleapis.com'

  #define GA_TEXT_REPLY
dialog_state_out.supplemental_display_text
  #define GA_AUDIO_REPLY  audio_out.audio_data
  #define GA_POSITION  latlng_pb2.LatLng

  /* 回答テキスト内の不要な部分を識別する */
  #define GA_UNNECESSARY  '\n--'

  /* 動作環境 */
  #ifndef MY_LANGUAGE
  #define MY_LANGUAGE  'ja-JP'    /* 使用する言語コード（日
本語） */
  #endif

  #define MY_DEADLINE  60 * 3 + 5  /* gRPC コールの完了期
限 */

  /* 設定内容 */
  #define MY_SAMPLE_RATE    16000  /* 音声サンプリングレー
ト 16 ksps */
  #define MY_INPUT_VOLUME    0  /* 入力音量 0% */
  #define MY_OUTPUT_VOLUME  100  /* 出力音量 100% */
  #define MY_AUDIO_IN    'ENCODING_UNSPECIFIED'
  #define MY_AUDIO_OUT    'MP3'
  #define MY_DISPLAY    'OFF'
  #define MY_DEBUG    False  /* デバグ情報は要求しない */
  #define GA_NEW_CONVERSATION  True  /* 初期値は新規会話
*/
  #define GA_CONVERSATION_STATE  None


  #define __GOOGLE

  #endif
```

9.9.6検索実行モジュール

Google が提供しているサンプルプログラムには、処理内容の説明がほとんどありません。「良いプログラムは、読むだけで処理が分かる」と言われていますが、汎用に作りすぎていることもあって、非常に分かりにくいものでした。不要な機能を削いで、手順を記述したのがこのモジュールです。順番は、

1. 認証を行って通信回線を確保し、Assistant オブジェクトを生成する（secure）
2. 要求内容を JSON として組み立てる（build）
3. Assistant に要求を送りつけ、応答（特に音声データ）を小出しに受け取る（request）
4. 音声を指定ファイルに収納し、テキスト出力を返す（askGoogle）

です。どこにも見つからなかった以下の情報を除いて、各々の処理の詳細説明は省略します。まず、緯度と経度を与えるには、latlng_pb2 というライブラリを使う必要があり、use-google.h でインポートと引用を定義しています。デバグ情報要求 debug_config が指定できませんでしたが、指定しなければ情報が得られないだけなので割愛しました。

askGoogle.py

```
  /* Google Assistant インターフェース
    初版： 2020/6/29
    最新版：

  簡略バージョン：一問一答で閉じる会話のみ
    リクエストはテキストを使って行う
    応答には音声とテキストの両方を期待する

  Class: GoogleAssistant

  属性：（簡易版では不要だが、対話を繰り返すように拡張するた
めに用意しておく）
    assistant:  Google Assistant オブジェクト（簡易版で
は毎回生成する）
    new_conv:  新規の対話フラグ（簡易版では常に真）
    state:  Google Assistant が返す会話状態。（簡易版では
常に空）

  操作：
    secure:  通信路を確立する
    build:  要求データを構築する
    request:  要求を実行する
    askGoogle:  Google Assistant に問い合わせ、返信から
データを得る

  */

  #include "include/use-google.h"

  class GoogleAssistant:
    def __init__(self):
      /* 簡易版では初期値を常に定数として使う */
      self.assistant = None
      self.new_conv = True
      self.state = None

    /* 認証を済ませ、通信回線を確立する */
    def secure(self):
      /* 認証ファイルを使って正規ユーザーである認証を得る */
      try:
        with open(MY_CREDENTIAL, READ_FILE) as f:
          credentials = GA_API_SECURE(f)
          refresher = GA_API_REFRESH()
          credentials.refresh(refresher)
      except Exception as e:
        print('Error in loading credintials: %s', e)
        return None
      /* 認証情報を使って回線を開く */
```

```
      channel = GA_API_CHANNEL(credentials,
refresher, GA_API_ENDPOINT)
      return GA_API_OPEN(channel)

    /* リクエスト情報を構築する --- 構造資料を参照すること
*/
    def build(self, query):
      req_conf = GA_API.AssistConfig(
        audio_out_config=GA_API.AudioOutConfig(
          encoding=MY_AUDIO_OUT,
          sample_rate_hertz=MY_SAMPLE_RATE,
          volume_percentage=MY_OUTPUT_VOLUME,
        ),
        screen_out_config=GA_API.ScreenOutConfig(
          screen_mode=MY_DISPLAY,
        ),
        dialog_state_in=GA_API.DialogStateIn(
          language_code=MY_LANGUAGE,
          conversation_state=self.state,
          device_location=GA_API.DeviceLocation(
            coordinates=GA_POSITION(
              latitude=MY_LATITUDE,
              longitude=MY_LONGITUDE,
            ),
          ),
          is_new_conversation=self.new_conv,
        ),
        device_config=GA_API.DeviceConfig(
          device_id=MY_DEVICE_ID,
          device_model_id=MY_DEVICE_MODEL,
        ),
  /* 情報不足のため削除……サポートされているか不明
        debug_config=GA_API.DebugConfig(
          return_debug_info=MY_DEBUG,
        ),
  */
        text_query=query,
      )
      return req_conf

    /* Google Assistant にリクエストを渡し、応答を少しづつ
受け取る */
    def request(self, myrequest):
      yield GA_API.AssistRequest(config=myrequest)

    /* Google Assistant に問い合わせて回答を音声とテキスト
で得る */
    def askGoogle(self, text, voice_file):
      reply = None
      ret = False
      /* 認証手続きを行い、Assistant オブジェクトを得る */
      self.assistant = self.secure()
      /* 上の手続きが成功したときのみリクエストを送る */
      if self.assistant:
        /* 要求を構築する */
        req = self.build(text)
        /* 音声応答を収めるファイルを開いて */
        with open(voice_file, WRITE_FILE) as f:
          /* こま切れの応答からデータを取り出す */
          for resp in
self.assistant.Assist(self.request(req),
MY_DEADLINE):
            /* テキスト応答があったら取り出す */
            if resp.GA_TEXT_REPLY:
              reply = resp.GA_TEXT_REPLY

            /* 音声応答をファイルに入れていく */
            if len(resp.GA_AUDIO_REPLY) > 0:
              f.write(resp.GA_AUDIO_REPLY)
        ret=True
      if reply:
        return ret, reply
      else:
        return ret, None
```

9.9.7検証

検証用のプログラムは掲載していないので、公開しているプロジェクトファイルを見てください。test_google_config.py を実行すれば、リクエストに使うデータのチェックができます。前のページにある AssistRequest と比較してください。

次に test_google_query.py を実行すると、実際に問い合わせが行えます。何も#define しなければ、現在地の天気予報を日本語で、 I_AM_IN_US を #define すれば、アメリカの都市からの移動時間を、I_AM_IN_HAGUE を#define すればヨーロッパの有名な美術館の所在地を（各国訛りの英語で）応えてくれます。返事は画面にテキストで表示されます（空の場合もある）。また、返事の音声は ~/voice/GoogleReply.mp3 です。sudo mpg123 で再生すると、各国語での返事が聞けます。質問が検索できなかった場合は、テキストは表示されず、音声で言い訳を言っています。

Google Assistant API は開発用に公開されているので、自分で付加価値のあるコンテンツを作り上げることを求めています。そのため、問い合わせの一部（特にニュースに関するもの）は、「ごめんなさい」としか答えてくれません。

9.9.8検索応答モジュール

検索応答モジュール use_cloud.py は、検索実行モジュールを呼び出し、応答をカオナシ Web と音声再生サブシステムに送ります。最初に「グーグルに応えさせます」と言い、音声応答を再生します。テキストから不要な情報「--- weather.com でもっとよく見る」を削除して、カオナシ Web に表示します。テキストが空の場合は表示しません。

インターネット接続ができていないなど、応答が得られない（success が偽）場合は、「ごめんなさい、応答がありませんでした」と応えます。

use_cloud.py

```
  /* Google Assistant インターフェース
    初版：2020/5/2 Chuji
    最新版：


  クラス：
    cloud_service
  属性：
    lip:     発話インターフェース
    fifo:     Redis FIFO
    ???
  操作：
    excuse:  グーグルに処理を任せると言う
    answer:  グーグルの返事を発話させる。テキストをカオナシ
Web に表示する
    no_text:  グーグルの返事を発話させる。カオナシ Web には
表示しない
    query_cloud:  問い合わせをする
  */

  #include "include/use-redis.h"
  #include "include/AI_commands.h"

  #include "askGoogle.py"
```

```
  class cloud_service:
    def __init__(self, mouth):
      self.lip = mouth
      self.fifo = open_FIFO()
      self.google = GoogleAssistant()

    def excuse(self):
      self.lip.ah()
      self.lip.say(WORD_GOOGLE)
      self.lip.speak_now()
      self.lip.reserve()

    def answer(self, reply, voicefile):
      self.lip.say_this(reply, voicefile)
      self.lip.ahah()

    def no_text(self, voicefile):
      self.lip.play_this(voicefile)
      self.lip.ahah()

    def query_cloud(self, question):
      self.excuse()
      sucess, reply =
self.google.askGoogle(question, VOICE_DIRECTORY +
MY_AUDIO)
      if sucess:
        if reply:
          if GA_UNNECESSARY in reply:
            reply =
reply[:reply.find(GA_UNNECESSARY)]
          self.answer(reply, MY_AUDIO)
        else:
          self.no_text(MY_AUDIO)
      else:
        self.answer(WORD_NO_REPLY, VOICE_NO_REPLY)
```

9.10　AI エンジン

音声処理モジュール、感情処理モジュール、時間管理モジュール、検索サブシステムについては、既に説明しました。

命令と返事の語彙をまとめた AI_commands.h にはだいぶ追加がありました。プロジェクトファイルを参照してください。

9.10.1挨拶モジュール

前章の「実証モデルで追加する機能の検討…挨拶の充実」にあげた定型挨拶を追加します。プロトタイプでは命令解釈モジュールで実現していた、簡単な挨拶も、このモジュールに移しました。ほとんどの定型挨拶は、パラメータを与えて `simple_reply` を呼び出しています。ここは関数マクロを使うべきところ（例えば

```
#define HELLO  simple_reply(WORD_HELLO)
```

と定義して、`self.lip.hello()` と呼び出す代わりに、`self.lip.HELLO` と書けばいい）ですが、大文字を使うと意図が伝わらない気がして、あえて関数として定義しました。

greeting.py は掲載しません。プロジェクトファイルを参照してください。検証は test_greeting.py を増強して行います。

9.10.2命令解釈モジュール

interpreter.py は掲載していません。プロジェクトファイルを参照してください。

シャットダウンの前にサブプロセスを終了させるため、命令解釈モジュールの初期化時にサブプロセスオブジェクトを渡すようにしました。

ほかには、プロトタイプで採用した、順番に言葉を確認していく方式のケースを増やしただけです。

命令ごとに感情処理モジュールを呼び出します。おだてればいい気分に、仕事をさせると不機嫌になるようにしました。

命令解釈モジュールで使っている、時間取得モジュールの改造については、時間管理モジュールのところで説明しました。

命令解釈で行うパターンマッチングは、設計段階で次の表のように整理しておきます。この表は PatternMatching.pdf としてプロジェクトファイルにも含めておきました。

この表ができていれば、条件ごとにロボットの応答や感情変化、具体的な動作を忠実に記述していくことができます。

命令	応答	感情	動作
（システム起動時）	（時刻による）おはようございます こんにちは こんばんは 今日は○○年○日、○曜日です	普通	-
（冒頭の）カナナン	-	-	ウェイクアップフラグを立てる
お休みフラグが立っているとき）	システムをシャットダウンします	最大の嫌悪感	システムをシャットダウンする
（いいよ）or（よろしい）or（はい）	さようなら		
（お休みフラグが立っているとき、上の命令がない）			お休みフラグを消す
こんにちは	こんにちは	好感	-
おはようございます）	おはようございます	好感	-
こんばんは	こんばんは	好感	-
ありがとう	どういたしまして	より好感	-
（ごくろう/さん）） or（ご苦労（さま）） or（よくやった）	ありがとうございます	最大の好感	-
（いい天気/だね）） or（良い天気/ですね））	そうですね	より好感	-
元気ですか）	おかげさまで、元気です	より好感	-
（起きて） or（おきて） or（ねえねえ）	はい	嫌悪感	再度ウェイクアップフラグを立てる
（疲れた） or（つかれた） or（眠い） or（ねむい）	元気を出してください	好感	-
（もっと）大きな声で話して）	このくらいの声でどうですか	好感	発話音量を上げる
（もう5ちょっと/小さな声で）） or（静かに）（話して）		嫌悪感	発話音量を下げる
（何か歌を）（歌って） or（唄って）	「いつも何度でも」を歌います	より好感	歌を再生する
（今日の天気） or（（天気）＆（（予報） or（調べて） or（教えて））	グーグルに答えさせます （グーグルの応答が続く）	嫌悪感	Googleに検索を要求し返答を言わせる
（ニュース）＆（（教えて） or（調べて） or（ある））	グーグルに答えさせます （グーグルの応答が続く）	嫌悪感	Googleに検索を要求し返答を言わせる
（（きょう） or（今日は））＆（何日）	今日は○○年○日、○曜日です	嫌悪感	（日付取得）
（いま） or（今））＆（何時）	いまの時○分です	嫌悪感	（時刻取得）
（（分） or（時））＆（なったら）＆（（教えて） or（知らせて））	○番タイマーを起動しました ○時○分にお知らせします （時間が来たら）○番タイマーの時間が来ました	より嫌悪感	（現在時刻を調べ、時間を計算する） ○番タイマーを起動する
（（分） or（時間））＆（（経ったら） or（たったら））＆（（教えて） or（知らせて））	○番タイマーを起動しました ○○分後にお知らせします （時間が来たら）○番タイマーの時間が来ました	より嫌悪感	○番タイマーを起動する
（（時間） or（時刻））＆（教えて）	いまの時○分です	嫌悪感	（時刻取得）
（（動画） or（ビデオ））＆（（とって） or（撮って） or（取って））	動画の撮影と配信を始めます	嫌悪感	動画の撮影を始め、停止命令を待つ
（（動画） or（ビデオ））＆（（とめて） or（止めて） or（停めて））	動画の撮影と配信を止めました	好感	動画の撮影を止める
（写真（を））＆（（とって） or（撮って） or（取って））	写真を撮りますよ、こっちを向いてください はい、チーズ （パチり） 写真を撮りました、ウエブで確認してください	好感	中精細度の写真を撮影する
（（きれいな） or（綺麗な））＆（写真（を））＆（（とって） or（撮って） or（取って））			高精細度の写真を撮影する
（写真）＆（メール）） or（（メール）＆（（送って） or（して））	メールを送信しました	嫌悪感	撮影した写真をメールに添付して送信する
（おやすみ（なさい）） or（お休み（なさい））	おやすみなさい システムをシャットダウンしてもいいですか	より嫌悪感	お休みフラグを立て、ウェイクアップワードがなくても命令を聞くようにする
（さようなら） or（さよなら）	システムをシャットダウンします	最大の嫌悪感	システムをシャットダウンする
シャットダウン（しなさい）	さようなら		
（上記以外の命令）	はい	嫌悪感	三回連続するまで、ウェイクアップフラグを消さない

パターンマッチング

このモジュールの設計段階で、思い付きを追加しました。すでに楽曲（mp3 ファイル）の演奏ができているので、一曲だけですが、「歌を歌って」と命令されれば「千と千尋の神隠し」の主題歌を歌うようにしました。SD カードの空き領域はかなりあるので、レパートリーを増やすことは可能です。

9.10.3AI エンジン冒頭部

AI エンジン冒頭部（kaonashi.py）は、プロトタイプに追加をしました。必要になるプロセス（タイマーを除く）をすべてここで起動します。マクロ定義 RUN_ALL（すべての子プロセスを起動する）と BCM2835（感情表現サブシステムでカラーLED を駆動する）は、通常#define します。デバグ途中で子プロセスがすでに走っていた時や、カラーLED を駆動しないときのみ#define せずに実行します。

シャットダウン命令を受けたときは、子プロセスにシグナルを送って停止させます。

9.10.4個別検証

ここまでで各サブシステム・モジュールの設計と個別検証を終わります。この本に掲載していないファイルは、別途ダウンロードできるプロジェクトファイルを参照してください。

9.10.5自動起動

実証モデルは、ロボットの電源投入で自動的に立ち上がるようにしましょう。/etc/rc.local の最後に以下に行を追加します。システムファイルなので、sudo で編集してください。

/etc/rc.local （末尾に以下を追加する）

```
（略）
/home/chuji/kaonashi/run_kaonashi
```

システムをリブートすれば、カオナシが自動的に起動されます。

コラム　球形の牛の先は？

「牛を球とした」あとは、どうなるのでしょうか（この話は『物理学者はマルがお好き』には書いてありません）？　おおざっぱに捉えたら、今度は少しずつ詳しく見ていくのです。牛の外形を表わす関数 f (x)を、次のような多項式で表現することを考えます。

$f(x) = a_0 + a_1 f_1(x) + a_2 f_2(x) + \ldots + a_n f_n(x) + \ldots$

この級数を適当なところで打ち切って、$f(x)$を「近似」するためには、後ろの方の項の影響が小さいことが必要です。第 3 項が第 2 項より（絶対値として）大きいことはあっても、それ以降がずっと小さければ、打ち切りは可能です。それは比較的分かりやすい、べき級数でも可能です。

$f(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n + \ldots$

次数の大きな項は、それだけ細かい形を表します。ところが、途中で打ち切ったべき級数では、最も精度よく $f(x)$を近似する $a_0, a_1, a_2, \cdots a_n$ の値が、打ち切った次数で変わってしまうという不都合があります。どこで打ち切っても、最適な係数 $a_0, a_1, a_2, \cdots a_n$ の値が一定（基底関数が直交するといいます）なら、まず定数 a_0（球！）で近似してから、ひとつずつ項を増やして精度を改善していくことができます。次のフーリエ級数がよい例です。

$f(x) = a_0 + a_1 \sin(kx+\theta_1) + a_2 \sin(2kx+\theta_2) + \cdots$

三次元の場合は、もう少し複雑な「ルジャンドル多項式」で近似展開できます。物理の世界では、これを「多重極展開」とも言います。各項は下のような関数で、それを合成したものが、「球」、「こけし」、「起上りこぼし」という風に、外形を少しずつ詳しく表現しています。

10　実証モデルの評価とフルモデル開発に向けて

実証モデルを動かして評価し、そこからフルモデル開発に向けた方向付けを行います。

10.1　実証モデルの評価

ここでは、次の点について、実証モデルを評価します。プロトタイプではパフォーマンス中心の評価でしたが、実証モデルでは使い勝手などの機能面も評価の対象にします。

- 立ち上がり、応答などの時間
- 機能の充足度
- 既存品（市販品）との比較

10.1.1応答パフォーマンス

まず、実証モデルの応答パフォーマンス（処理に必要な時間）を計測します。カオナシ立ち上げから始業挨拶までと、語りかけ（命令）から応答までの時間を、同一条件で調べた結果を下の表に示します。「内部処理」は Raspberry Pi ZERO 上で Julius を動作させた場合です。

Julius サーバー / 項目	応答時間（秒） 内部処理	サーバー1	サーバー2
kaonashi 起動から始業挨拶まで	38	30	30
音声命令への定型返答が Web に表示されるまで	28	3	1
音声命令への定型返答が発話されるまで	28	3	1
日付や時刻を訊いてから Web に表示されるまで	28	5	1
日付や時刻を訊いてから返答が発話されるまで	32	9	6
天気予報の問合せから Web に表示されるまで	38	5	2
天気予報の問合せから返答が発話されるまで	40	5	2

Julius サーバー毎の応答パフォーマンス

語りかけ後は 0.5 秒の無音時間を検出してから処理を始めるので、1 秒程度なら「ほぼ瞬時」と言っても差し支えありません。サーバーのスペックは以下のとおりで、それほど高性能でもありません。

サーバー	主要スペック
内部処理	ARM1176 (32bit,1GHz, 1 core, 128kB cache), 512MB memory, 16GB SD card
サーバー1	Atom N270 (32bit, 1.6GHz, 1 core, 512kB cache), 1GB memory, 32GB SSD
サーバー2	Celeron B840 (64bit, 1.9GHz, 2 core, 2MB cache, 4GB memory, 320GB HDD

サーバーのスペック比較

Julius は 2 コアで処理する設計になっているそうなので、64 ビット複数コアの CPU であれば、実用上充分だと思います。

周囲ノイズ（町の騒音や部屋のテレビの音など）が大きいときは、それを拾って、次々と Julius に送りつけるため、極端に応答が悪くなります。ノイズを減らしたうえで、入力音量を下げ、ロボットの近くで話しかけることで、大幅に改善できます。これで、一番高コストだった音声認識の改善ができました。

二番目に高コストだった音声合成の問題は、まだ残っています。これも Raspberry Pi ZERO の外に出した方が良さそうです。天気予報の音声ファイルをもらってきても、伝送時間は、ほとんど気にならないので。

10.1.2機能の充足度

実証モデルの機能は、「必要なものを取り上げる」というより、「実現が面倒なものは後回しにする」という方針で設定しました。商品開発のときは許されないことですが、仕様が明確に描けず、勉強を兼ねた研究開発型のアプローチを取らざるを得なかったからです。それで、実証モデルの機能は十分なものだったのでしょうか？

会話の種類はそれなりに増えています。時間管理や天気予報は、けっこう便利です。いっぽう、話しかけた人の認識や人物情報の管理は切り捨ててしまいましたが、自分ひとり用なら気になりません。スマートスピーカーも使用者の認識はしないので。

ここまでは主観的な分析なので、他のロボットと比較してみることにします。

10.1.3既存品との比較

機能の充足度を評価するときは、既存（市販）品と比較してみるのが有効です。いずれスマートスピーカーの機能はほぼカバーするつもりなので、開発構想のヒントになった他の 2 種類のロボットと比較してみます。

コミュニケーションロボット

最初の比較対象は「癒し系」のコミュニケーションロボットです。取り上げたのは、2020 年に発売されたばかりで、いちばん機能が充実している「お

しゃべりけんちゃん（前ページの写真）」です。話しかけると、さまざまな返事をするというものです。もう少し前からある、別のメーカーの「くまの子くーちゃん」と一緒に比較します。ただし実物を見たわけではないので、ネット上の広告だけを頼りにしています。下に比較表を載せました。

市販品はさすがにサービスが行き届いていて、主要ターゲット顧客である高齢者に「カワイイ！」と言わせる工夫がされています。第一に、本物の子どもの声の録音を再生しています。カオナシの声は暗いし、抑揚もおかしいので比較になりません。しかし声色は開発構想の違いなので優劣はつけられないと思います。また秘書は相手を「おじいちゃん」とは呼びません。この 2 点は比較の対象にしません。

定型の応答の種類はほぼ同程度ですが、既存品では人間臭さを出すため、ELIZA と同様、ランダムな揺らぎを作っています。歌のレパートリーも豊富なので、フルモデルで検討しても良いかもしれません。もっとも、スマートスピーカーなら、レパートリーは広くできます。

お遊びやひとり言は、面白い工夫だと思います。ただこれは、子どもだからカワイイのであって、妖怪がやったら不気味かもしれませんね。

コラム　暗いロボット

妖怪カオナシはちょっと暗い性格です。礼儀正しく丁寧というロボットばかりではなく、こういった「根暗なロボット」も小説などによく出てきます。

私がいちばん好きなのは、ダグラス・アダムズの『銀河ヒッチハイク・ガイド』に出てくる、パラノイド・ロボットのマービンです。2005 年の映画化（右）では、あの名優アラン・リックマン（『ハリーポッター』で魔法薬学教授のスネイプ先生を好演）が声を演じました。この根暗さが堪らない……。

石川藤夫の『ブラックホール惑星』などに出てくる、アールも暗い過去を持っています。そういえば、アールは堂々とした錆々声で歌っていました。不完全な良心回路で悩む『人造人間キカイダー』（石ノ森章太郎原作）や『鉄腕アトム：地上最大のロボット』に出てくるプルートゥも、悩みを抱えています。

項目	おしゃべりけんちゃん	くまの子くーちゃん	秘書ロボット　カオナシ
外形・イメージ	6 歳の男の子	こぐまのぬいぐるみ	妖怪
想定する使用者	子どもが独立した高齢者	高齢者、（子ども）	？？？
使用者への呼びかけ	「おじいちゃん」など 10 種類	「おじいちゃん」など 8 種類	自分からは呼びかけない
発話音声	本物の男の子の声	（不明）	合成音声（暗い声）
認識する語彙	28 語	20 語	28 パターン（33 語）
定型応答のパターン数	（不明）	100 通り（組み合わせ多数）	定型は 28 通り
歌を歌う	66 曲（ランダム選択・季節依存）	50 曲	1 曲
お遊び	なぞなぞを問いかける		
	占いをしてくれる		
	体操の掛け声をかける		
日付確認	日付や曜日を答える		日付と曜日を答える
アラーム	指定時間に目覚ましの声かけ		指定時間に声かけ
答えの確認	「もう一回言って」で答えを繰り返す		
その他	会話をしないと、時々ひとり言（370 パターン）を言う	昭和時代の話題が中心	アラーム、天気予報、……
動作を開始する操作	背中をトントンする	背中スイッチ、手を握って振る	電源スイッチ、音声、Web
電源	単三電池 4 本	単三電池 3 本	DC5V

癒し系コミュニケーションロボットとの比較

テレプレゼンスロボット

もう一つの比較対象として、代表的なテレプレゼンスロボットである Orihime を取り上げます。右は既に供給が始まっている身長 23 センチのモデルで、代理で会議に出席したりする用途に使われています。この外に、身長 120 センチで、動き回ったり物を運んだりできるモデルもありますが、開発構想が異なるので、比較しません。

可動部がある点が特徴で、とくに能面をイメージした顔は、角度で感情を表現できるとしています。すばらしい発想ですね。カラーLED の点滅くらいでは追いつきません。それ以外にハードウエア面での違いは少ないと思います。

カオナシは会議開催も目指しているので、参考になります。とくに顔（カメラ）の向きを変えられる点がいいです。カオナシの音声入出力を切り替えれば、いまでもテレプレゼンス機能は実現できます。代理される（リモートにいる）方は、動画が表示されるブラウザの画面を見ながら会話すればいいわけです。一番気になるのは、外部からカオナシの Web サーバーにアクセスできるようにすると、「覗き見」が可能になってしまうという点です。お年寄りの「見守り」にも使えると思いますが、セキュリティの設定は慎重にしたいものです。

項目	OriHime	カオナシ
用途	テレワーク 障がい者の代行	秘書 （見守り？）
外形	未来型	妖怪
素材	プラスチック	ぬいぐるみ
可動部	両腕と頭	なし
カメラ	1 基	1 基
音声入出力	マイク・スピーカ	マイク・スピーカ
音声の用途	現場との通話	命令と応答
感情表現	顔の角度	カラーLED アレイ
通信機能	WiFi	WiFi
画像表示	ホスト PC	ブラウザ

テレプレゼンスロボット Orihime との比較

10.2　フルモデルの開発にむけて

以上の評価を踏まえて、フルモデルの開発に向けた方針を検討します。

10.2.1高性能サーバーへの移植

Raspberry Pi ZERO の非力さは如何ともしがたいので、大部分の処理をより高性能のサーバーに移植しようと思います。ロボット本体内の Raspberry Pi ZERO は低消費電力である特徴を生かして、ハードウエアの駆動機能だけに限定し、処理負荷の重いサブシステム（あるいはモジュール）はサーバーに処理させます。そのとき考慮するのは、

- ハードウエアを操作する機能は Raspberry Pi ZERO に残す
- Raspberry Pi ZERO とサーバー間の通信量を増やさない
- 一方だけが稼働しているとき異常動作をしない

ことです。

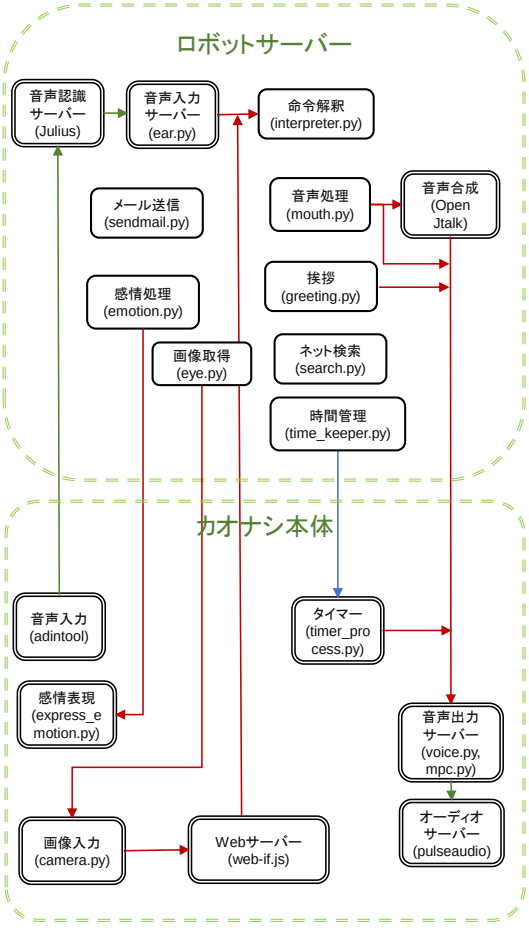
コラム　プロジェクト･ゼロ

秘書ロボットのフルモデルを検討しているとき、思い出した SF 小説があります。石川英輔の『プロジェクト・ゼロ（早川書房：1984）』は、日本企業のエンジニアたちが、まじめに性産業用ロボットを開発する物語です。メカ（？）はそれなりにでき上がったのですが、ロボットの言動を目的どおりに仕上げるのに苦戦しました。有史以来の膨大な指南書をもとに対話・行動システムを構築したのですが、そのために必要なコンピュータはロボット本体より大きなものになってしまいます。苦肉の策として「仕事部屋」隣の空間に設置し、無線でロボットを制御することになりました。ロボット本体にはセンサとメカが収納されています。

35 年前の発想を、それと知らずに借用していたんですね。もっとも石川氏は、写真と印刷に関わる技術者でもあったので、ごく自然に考えついたのではないかと思います。最新技術である、無線を含む高速通信網とスーパーコンピュータを使えば、隣の部屋にサーバーを置かなくても、このロボットが（特殊なメカは別にして）実現できるかもしれません。

サブシステム間の通信を単純にした Redis FIFO は、リモートでも使えるので、IP アドレスの異なる Redis サーバーを使い分けるだけです。マルチプロセッサシステムとしての課題である、相手が動作していない場合の処理も大事です。自分の中で動いている Redis サーバーへの blpop は問題ありませんが、相手の Redis サーバーへのアクセスは、サーバーが動いていないときの処理を組み込む必要があります。

Web サーバーの処理はあまり重くないので、Raspberry Pi ZERO が処理しても良いと思います。結局、下図のような構成が考えられます。



フルモデルの構成（拡張前）

10.2.2インクリメンタル拡張

フルモデルに搭載したい機能は多岐にわたり、比較的簡単に実現できるものと、調べたり試したりする必要があるものが混じっています。まとめて実現するより、順次追加していくことを選びます。そうすれば、秘書ロボットが「だんだんに進化していく」ことが実感できると思うからです。

新しい機能は、新しいサブシステム、あるいは新しいモジュールの追加や入れ替えで実現していきます。そのとき、左の構成図が指針になります。

10.2.3ルールエンジンの実現

実証モデルまでは、命令に含まれるキーワードのパターンマッチングで AI 機能を実装していました。解釈する命令が増えてくると、パターン間の干渉が発生して、if ～ elif ～で実装するのが、ますます困難になってきます。そろそろルールエンジンが必要になると思っています。

LISP で書き直すか、Python のメタプログラミングを行うか、durable_rules のような既存 Python フレームワークを使うか、まだ決めかねています。いずれにしろ、勉強することが多いので楽しみではあります。

10.3　フルモデルで拡張したい機能

前々節の機能評価をもとに、フルモデルで拡張したい機能をまず考えてみましょう。

10.3.1会話履歴で変化する応答

実証モデルでの会話は一問一答方式で、応答した後は（感情変数を除いて）忘れてしまいます。会話履歴を覚えていると、会話が豊富にできます。

例えば、「パスタを茹でているから、7 分経ったら教えて」とか、「3 時に山田さんに電話するから知らせて」と言われても、いまは「1 番タイマーの時間が来ました」としか報告しません。そのあとに「パスタをあげてください」とか「山田さんに電話してください」と言えると、ずっと秘書らしく、役に立つ会話になります。

そのためには、命令から意味をくみ取る必要があります。AI の本丸のような機能ですが、必ずしも意味を「理解する」必要はありません。ちょと人間の「常識」を振りかけるだけでもいいはずです。タイマーが必要になるのは、多くの場合、調理や作業をしているときです。アラームは、テレビのスイッチを入れたり、誰かに連絡したりするときに使いそうです。関連するキーワードを見つけることができれば、（多少あてずっぽうになるでしょうが）応答させることは可能です。

10.3.2応答の確認

コミュニケーションの基本である、聞き直しも取り入れたいと思います。最後の応答を覚えていて、もう一度言わせる機能です。

ロボットの「誤解」を解く機能も必要です。「1 分後に知らせて」を「10 分後」と聞き間違えていたら、訂正する仕組みが欲しいですね。

10.3.3雑談

せっかくのコミュニケーション機能なので、業務以外の雑談ができると良いと思います。日本人の（あたりさわりのない）雑談では、天気の関する話題が多く登場します。いまは「そうですね」くらいしか答えられませんが、もっと豊かにできるヒントがあります。

環境データの測定がそれです。ボッシュ社の BME280 というチップを使ったセンサモジュールが秋月電子通商から発売されています。I2C バスで通信でき、温度と湿度、それに気圧が測定できます。日付と組み合わせれば、「だいぶ暖かくなりましたね」といった季節の挨拶ができます。気温と湿度から不快指数を計算して、「蒸し暑くてたまらないですね」と言わせることもできるし、「気圧が下がってきたので、天気が悪くなりそうです」といった会話も可能です。もちろん、Raspberry Pi 定番の温度センサだけでも役に立つと思います。

実は、ハードウエアとしての環境センサは既に組み込んであるので、ソフトウエアを追加すれば使える状態です。補正演算の詳しい資料がない（C 言語のライブラリしか提供されていない）ので、Python で書いたドライバーソフト（environment.py と bme280.h）、それにモニタリング用のプロセス（env_monitor.py）をプロジェクトファイルと一緒に公開しました。Web サーバーと HTML 文書には、モニタリングしたデータを表示する拡張ができているので、プロセスを走らせればカオナシ Web 画面でデータが見えるようになります。

```
$ cpp env_monitor.py |sudo python
```

ドライバーソフトは、フルモデルで AI エンジンが環境データを読むときのために用意しました（その時には env_monitor.py は使用しません）。測定値を読みこんでいる間に、チップの測定部が値を更新してしまうのを避けるため、8 バイトのブロックを一度に読み出しています。

10.3.4隠し芸の充実

ロボットに歌を歌わせることはできているので、芸の範囲を広げることを検討します。「おしゃべりけんちゃん」のような童謡や、ソプラノを歌わせるのは、カオナシの雰囲気に合いませんね。

そこで落語を語らせてみました。あまり面白くなさそうな顔で滑稽噺を演じる、（十代目）柳家小三治の録音（著作物なのでプロジェクトの公開ファイルには含まれていません）をやらせることにしました。持ちネタからランダムに選んで口演します。このファイルは Raspberry Pi ZERO 上に置いておけます。

私自身はあまり好きではないのですが、占いをやらせることは可能です。簡単なプログラム例が公開されているので、好みのものを移植すればいいでしょう。

10.3.5気まぐれ行動

ロボットの応答がいつも同じでない方が「人間味がある」という意見もあるので、この点の検討をしておきます。ELIZA のように、乱数を使って複数の可能な応答から一つを選ぶ手法があります。応答そのものだけでなく、語尾を少し変えるのも有効です。

「おしゃべりけんちゃん」のところで述べたように、「飽きたときに」勝手な行動をとるのは、あまい秘書らしくありません（人間らしくはある）。それでも実装方法だけは考えておきます。機器やシステムが暴走したときに強制リスタートさせる、Watch dog timer という仕組みがあります。適当な頻度で犬の頭をなでているうちは何もしませんが、設定した時間以上やらないと、「番犬が吠え」るというものです。これを簡素化した仕組み（プロセス）を作って、「気まぐれ行動」を起動すればいいわけです。カメラで動画を撮影したときの仕組みが応用できます。

AI エンジンには、命令による動作をするたびに、そのプロセス用の Redis FIFO に信号を送らせます。プロセスは時間を測るためのカウンター変数を 0 に初期化したら、一定時間（例えば 1 秒）毎に、以下の動作をさせます。

- カウンターを 1 だけ進める
- Redis FIFO に信号が来ていたらカウンターを 0 にリセットする
- カウンターの値が指定しきい値を越えたら、（REDIS_TO_AI に命令を送って）AI エンジンに気まぐれ動作をさせる（カウンターはリセットする）

カウンターのしきい値を乱数で変化させると、より予測しにくい動作が期待できます。

10.3.6検索機能の拡張

実証モデルの天気予報では、要求としてあらかじめ用意したテキストを Google Assistant に送りつけました。Julius の日本語認識が多少おかしくても、検索が行えるようにするためです。

認識精度が上がってきたら、AI エンジンのパターンマッチングにあたらなかったが、それなりの長さのあるテキストは、（パターンマッチングでカバーできなかった命令を含んでいると想定して）そのまま Google Assistant に与えてもよいと思います。これでスマートスピーカーらしくなります。ただし、Google Assistant は開発用なので、ニュースや音楽再生など、付加価値の高いサービスには応じてくれません。別の（有料）サービスを考えた方が良さそうです。

一方、返事にカオナシの声と Google Assistant の声が混在するのは、興ざめな気もします。Google Assistant のレスポンスは、返事の音声をテキストとしても返すので、カオナシ Web に表示しました。このテキストを Open Jtalk でカオナシの声にすれば、カオナシが一人で全ての業務をこなしている印象が与えられそうです。天気予報だけなら、それも可能です。しかし、テキストが返ってこないとき（検証に使った、英語の道順や美術館の場所など）もあるので、全面採用は諦めました。

10.3.7感情表現

カラーLED による感情表現は、いまひとつピンとこないものがあります。もっと直接的に表現しようと思ったら、人形の顔の部分にカラー液晶を埋め込んで、表情のある顔の画像を表示するのが良いと思います。ビデオ会議で相手の顔を表示するのにも使えそうです。

しかし簡単に手に入る液晶パネルだと、画面が大きすぎるので、ロボット全体を大型にせざるを得なくなってしまいます。ガラケーに使われていたような 3～4 インチのパネルが手に入ったら考えようと思います。

コラム　C-3PO のお祖母さん

1927 年のドイツ映画『メトロポリス』（1948 年の手塚治虫作品とは別物）では、階級社会で労働者を団結させようとした女性マリアに似せたロボットが、人々を扇動するという物語です。マリアの擬態を剥がされたロボットは（白黒映画ですが、たぶん）金色に輝き……、のちに C-3PO のデザインの元になったそうです。

10.4　フルモデルで新たに実装する機能

実証モデルにはなかった構想の機能を検討します。これも、ひとつずつ実現していきたいです。

10.4.1予定管理

予定管理は重要な秘書の業務です。私は PC、タブレット、スマートフォンなどのアプリを使っていますが、その全部を Google Calendar に同期させて、いつでもどこでも予定を調べたり、作成したりできるようにしています。ロボットでもできるといいですね、特に口頭で指示できれば。

スマートスピーカーには、この機能が組み込まれています。Google Assistant は Google Calendar との相性がいいので、組み込んでみようと思います。

10.4.2メールの送受信

定型メールの送信は実証モデルに組み込みました。それ以外のメールの作成と送信、それに受信について考えてみます。

住所録に相当するデータベースは必要そうです。名前（読み方付き）とメールアドレスの辞書を用意して、ファイルにできるようにします。

受信したメールをサーバーから取ってくるのには、サーバーに受信メールを残しておける IMAP プロトコルを使います。メールを印刷するプログラム例を参考に、送信者（辞書で名前に直しておきます）、件名、本文を発話させます。引用部分は読まないとか、確認のため読み直しをするといった工夫をすると良いと思います。

オリジナルメールの送信は少し面倒です。件名と本文を用意するには、話した内容をテキストにする、「口述筆記」という機能が必要だからです。結果を秘書に読み上げさせて、聞き間違いを訂正したり、文章を推敲したりすることも求められます。キーボードを叩いた方が簡単で速いのかもしれません。口述筆記は、研究テーマとして、じっくり取り組んだ方が良さそうです。

10.4.3音声・画像通信

秘書ロボットを使って外部と通信する機能について、ハンズフリーの電話機から、ビデオ会議まで考えてみます。

音声入力の横取り

クラウドサービスを検討するときにも問題になりましたが、マイクからの音声入力は Julius が使ってし

まいます。Julius 以外のソフトウエアが音声入力を使えるようにする工夫が必要です。

Julius/adintool の仕様をよく読んでみると、マイクやファイル、adinnet 以外の入力手段がありました。Linux の標準入力がそれです。例えば、arecord の録音データを標準出力に繋ぎ、それをパイプで Julius に与えればいいわけです。パイプなら、2 分岐コマンド tee で録音データが取り出せます。簡単なプロセスを自作して、ロボットが話してるときは（混信を避けるため）音声入力を止めることも可能です。

音声通信

HTML5 以降のブラウザ（例えば Chrome）では、音声ファイルの作成や再生ができます。これを使えば、ロボット側とブラウザ側で音声通信ができるようになります。実証モデルで作ったカオナシ Web とは別の Web サーバー（別のポート番号を持つ）と HTML 文書を用意します。インターネットとの間にあるファイアウォールの設定を変更し、このポートだけはロボットに繋ぐようにします。Web サーバーにはログイン手順など、厳重なセキュリティを設定します。

ブラウザに表示させたオフ・フックボタンを押したら、「○○さんから電話です」と言わせ、許可したら双方向通信できるようにします。Julius の機能は生かしておいて、会話内で「カオナシ、電話を切って」と言えば終了できるようにできます。

相手がブラウザを開いているかどうかわからないので、ロボット側から通信を始めることはできません。いまのところ、定型メールで「電話ください」と送る（昔のポケットベル相当）しかありません。

見守り用途

一人暮らしのお年寄りを見守るには、相手側のブラウザに動画も表示するといいです。環境データを表示すれば、エアコンを適切に使っているかどうかも確認できます。前にも述べましたが、覗き見対策は重要です。

テレプレゼンス

ロボット側の映像が見えて、双方向通話ができれば、テレプレゼンス媒体としても使えます。会議に代理出席したり、旅行に同行させたり。カオナシの外観から想像を広げて、占いや人生相談窓口に置くのもいいですね。

ビデオ会議

テレプレゼンスに、相手側の画像も見えるようにすれば、ビデオ会議になります。最大の障害は、ロボット側に画像を表示するハードウエアがないことです。カオナシ Web に表示して、タブレットをそばに置くか、ロボットの顔に表示器を埋め込む必要があります。

このくらいになると HTML5 ではなく、リアルタイム通信用の WebRTC を使った方がいいと思います。

10.4.4話相手の認識

何もしていないカオナシに話しかけたとき、その人の画像を取り込んで、誰なのか認識できれば、会話の幅が広がります。

最近話題になっている OpenCV は、Raspberry Pi にも実装できます。それで「顔が認識できた」というのは、顔を見つけられたということだけで、誰かまでは分からないようです。

画像に写っているのが誰なのか推定するクラウドサービスとしては、次の二つが有名です。

- Microsoft Cognitive Service の Face
- Amazon Web Service の Rekognition

前者は Web サイトである程度実験できるので、こちらから試してみようと思います。

10.4.5感情の読み取り

当初は、顔写真から感情を読み取るクラウドサービスを使って、「嬉しそうですね」とか「なにか心配事でもあるのですか」といった会話をさせることを目論んでいました。

しかし実際に写真を使って試してみると、日本人は感情を外に出すのが苦手なのか、ほとんどの顔が「嬉しい」か「中立」と判定されてしまいました。欧米人の顔で学習してきたせいかもしれません。すでに応答は作ってあるので、実装は難しくなさそうですが、実用性（と言うより、面白い会話になるか）は、何とも言えません。

候補になるクラウドサービスは顔認識と同じです。

10.4.6機器操作

この本の冒頭で上げた、秘書ロボットの業務・作業は、ここまでの検討でほぼカバーしました。残っているのが、機器の操作です。音声で照明や空調、音響機器の操作をするものは、スマートリモコンなど

と呼ばれています。重要な検討事項は、その機器をどうやって（物理的に）操作するかということです。考えつくものを挙げたのが下の表です。

機器	操作手段
エアコン	赤外線リモコン
電気ストーブ	AC スイッチ
扇風機	AC スイッチ/赤外線リモコン
換気扇	AC スイッチ
テレビ・ビデオ	赤外線リモコン
オーディオアンプ・ラジオ	赤外線リモコン
録音機器	AC スイッチ/タッチスイッチ
PC	電源（接点）スイッチ
天井灯	AC スイッチ/赤外線リモコン
移動できる照明機器	AC スイッチ/赤外線リモコン

スマートリモコンで操作する手段

いちばん単純なのが、AC 電源をそのままオン・オフすることです（危険を伴うので、組み立てに気をつける必要がありますが）。前のプロジェクトで電熱器を半導体リレーで制御した方法がそのまま使えます。しかし、秘書ロボットにコンセントを装備して、電源操作をするのは気に入りません。言葉を交わせる場所の制約もあるし、ロボットからあちこちに電源ケーブルが伸びるのもスマートでないからです。WiFi 経由でオン・オフができる、スマートプラグを用意した方がいいですね。

Raspberry Pi を赤外線リモコンにする試みは多くあるので、実行可能だと思います。ただし、リモコンの信号（フォーマットと言います）は何種類もあり、メーカー毎に異なるので、実際の信号を調べたりする必要があるようです。受信装置の方を向けないと動かないことがあるため、これもスマートプラグのように、別の固定デバイスにした方がいいかもしれません。

10.4.7やっぱり C-3PO

前にも書きましたが、標準声色のピッチや速度を変えても、クラウドが合成した音声では、あまりカオナシの雰囲気が出ません。もっと秘書らしい声にして、外観も変えていくことも考えています。今回の加工では人形を台座に載せました。エレクトロニクスやスピーカーを、もっと大きな台座に収納すれば、小型の C-3PO でも良いと思います。眼の裏にカラーLED チップを、胴体（腹部の黒い部分？）にカメラを埋め込めば、それほど違和感はないと思います。

秘書ロボットとしての C-3PO の素材を探していたら、Jakks Pacific 社の身長 80cm 近いプラモデルを見つけました。50cm のモデルもありましたが、今はどちらも作られておらず、入手は簡単ではありません。このサイズなら、内部にかなりの電子回路が収められるので、探してみようと思います。黄土色のプラスチック製なので、写真のように金ぴかに塗装すると良いそうです。

10.4.8可動型ロボット

ここまで封印してきた、可動部のあるロボットも検討したいと思います。すぐに動作する素材は Rapiro です。Raspberry Pi A または B を内蔵できる設計になっていますが、Raspberry Pi ZERO で空きスペースを広くとれれば、オーディオユニットやスピーカーを入れられそうです。額にカメラ用の窓もあります。可動部の制御は専用基板（Arduino 搭載）が行うので、Raspberry Pi ZERO の負荷は増えません。

プラスチック部品を少し加工すれば Raspberry Pi B+も搭載できるので、クラウド以外の外部サーバーを使わないロボットができるかもしれません。ただし、単三電池で全体を動作させるのは難しそうですね。歩き回る時には、電源ケーブルを引きずりたくありません。

もう一つの可能性は、ロボットを車両に載せてしまうという選択肢です。大八車のようなキットが簡単に手に入るので、それに載せれば自由に動き回れます。モーター制御や、衝突回避用のセンサ入力などが必要になりますが、これは Raspberry Pi の工作でもよく出てくるテーマです。電池駆動を前提にすると、どこまで省電力化できるかが、カギになりそうです。写真のようなイメージですが、これはもう秘書ロボットではありませんね。次のプロジェクトとしては良いテーマです。ラジコンカーではなく、自動運転車（あるいはプログラムで動く車両）を目指したらどうかと思います。

11　プロジェクトを振り返って

Raspberry Pi を使った秘書ロボット開発プロジェクトは、実証モデルができ上ったので、この本の説明は終了とします。じゅうぶん楽しめる機能は実現できました。

11.1　実証モデルまでの成果

実証モデルの開発まで行って、いろいろ習得できたことがあります。それは、私にとって重要な順番でいうと、

- トップダウン一辺倒でない、研究開発型アプローチ
- システムを複数のプロセスから構成する設計の考え方
- プロセスをサーバーに分散配置することで処理能力を向上させること
- モジュールを交換することで、プロセスの機能拡張や変更ができるようにする工夫
- プロセス間インターフェースの多くを Redis に統一し、検証やモジュール交換を容易にする手法の確立
- クラウドサービスをデバイス（ここでは秘書ロボット）から利用できるようにすること
- 規則型 AI エンジン構築の経験
- 簡単な Web サーバーの構築ノウハウ
- 他の人が開発したソフトウエアを常駐プロセスに改造すること
- 視覚、聴覚、発話、感情表現の組み込み経験

といったことです。

現在は、フルモデルのために検討した機能を、一つずつ作りこんでいるところです。全体の完成がいつになるか分からないので、ここまでを公開することにしました。

カオナシ プロトタイプ

実証モデル

フルモデル

11.2　常識外れ（？）のアプローチ

この本のはじめで、『物理学者はマルがお好き』の示唆をまとめました。それは、

A. 牛をまず球とみなせ（複雑なものは、枝葉末節を切り捨て、簡略化して考えよう）
B. 落とし物は街灯の下から探せ（できる可能性のあるところから手をつけよう）
C. うまく行ったら、また同じようにやれ（検証されたものは、何度でも使いまわそう）

あるいは、今回のプロジェクトについて読み替えて

A. いきなり複雑なシステムにしない（開発フェーズを設定し、発展させていく）
B. 実現に時間がかかりそうなものは後回し（簡単なお試しだけにしたり、延期したりする）
C. 実績のある手法は繰り返し使う（温度コントローラで役立った手法を再利用する）

でした。A.について、コラムでも述べた近似で考えれば、今回のプロジェクトのフェーズは、左下のイメージです。だんだん目標に近付いているのが分かりますか？

B.について、「落とし物を街灯の下で探す」ことの愚は、朝永振一郎先生の随筆で学んでいたのですが、そこには「その下を歩いた覚えがない」という前提があることを忘れていました。「安易な道」というのではなく、「うまくいく可能性の高い道」があるのなら、そこを行くべきだと思います。

C.については、前の温度コントローラ・プロジェクトで身につけた、Web サーバーや Redis FIFO、それに cpp を利用した保守の容易化が、今回も使えました。今回身につけたマルチプロセス・マルチプロセッサ化や、クラウドの利用方法は、この次のプロジェクトでも活用していこうと思います。

『物理学者はマルがお好き』は、まじめな啓蒙書なのですが、著者のクラウス先生は、私のような都合のいい読み方をする読者でも、「面白い、楽しい」と思うなら、歓迎してくれる人のようです。示唆に富んだ著書に感謝します。

Linux と Windows のファイル

先に Windows と Linux の間のファイル互換性問題を指摘しました。もう一度整理すると

1. 日本語コードが異なる。Linux では UTF-8 を、Windows では Shift-JIS を使っており、互いの日本語を表示できないことがある。
2. 一行の終わりのコードが異なる。Linux では newline（NL）一文字、Windows では Carriage-Return（CR）と Line-Feed （LF）の二文字を使う。

というものでした。この間の変換を行うツールとして Linux の nkf（Network Kanji Filter）というものがあります。Raspberry Pi や Ubuntu PC にインストールする手順は下のとおりです。

```
$ sudo apt-get install nkf
```

nkf にはいろいろオプションがありますが、とりあえず次の 6 つだけ知っていれば役に立ちます。

オプション	処理
--guess	ファイルの日本語コードと改行を表示する
--overwrite	変換した結果を、元ファイルに上書きする
-s	Shift-JIS に変換する
-w	UTF-8 に変換する
-Lw	改行を Windows 式（CR+LF）に変換する
-Lu	改行を Linux 式（NL）に変換する

nkf のオプション（一部）

--overwrite を指定しなければ、変換結果は標準出力に表示されるので、リダイレクトで別のディレクトリや別ファイルにすることもできます。

私のプログラムは区切りにタブを使うことが多いのですが、この本にプログラムリストを掲載すると横幅を取りすぎます。それでスペース 2 文字に置き換えるプログラム t2s.c を用意しました。

t2s.c

```
/* tool to replace a TAB with two SPACEs */

#include <stdio.h>

#define NL      '\n'
#define CR      '\r'
#define LF      '\n'
#define TAB     '\t'
#define SPACE   ' '

int c;

int main(){
        while ((c = getchar()) != EOF){
                if (c == TAB){
                        putchar(SPACE);
putchar(SPACE);
                }
                else putchar(c);
        };
}
```

使うコンピュータ上でコンパイルして使います。

```
$ cc -ot2s t2s.c
```

これをホームディレクトリ（~/）に置いて、以下のシェルスクリプト port2dos を用意しました。一番目のパラメータで指定した拡張子のファイルを変換して、二番目のパラメータで指定したディレクトリに同名で格納します。

port2dos

```
# tool to convert UTF-8+newline to SJIS+CRLF and
replace TAB with two SPACEs
# usage: port2dos file-extention directory-to-save

for f in *.$1;
do
        nkf -s -Lw $f | ~/t2s >$2/$f
done
```

下の使用例では、カレントディレクトリにある*.py ファイルをすべて、Windows 形式にしてから、タブをスペースに変換して、結果を../dos に収納します。

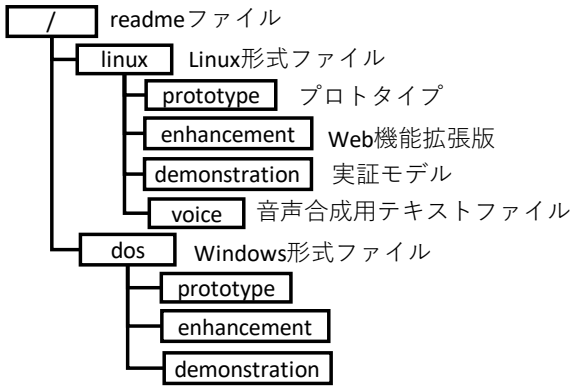
```
$ port2dos py ../dos
```

プロジェクトファイル

このプロジェクトで作成したファイルをまとめてダウンロードできるようにしてあります。学習や研究目的であれば、自由に改造や再配布をしてもらって構いません。ただし著者は、プログラムの実行によって起こった、どのような事態にも責任は負いません。

私が使っているサーバーやサービスのアカウントに関する情報などは、配布ファイルから一部を削除してあります。ファイルを皆さんのプロジェクトに利用するときは、この部分に皆さんの情報を追記したり、適宜修正したりして使ってください。

ファイルは ZIP 圧縮されており、以下のようなディレクトリ構成になっています。linux ディレクトリには、そのまま Raspberry Pi で使えるファイルが、dos ディレクトリには、前出のシェルスクリプト port2dos で変換したファイル（タブを空白で置換しているので、環境によっては、そのままで使えないときがあります）が収納されています。t2s.c や port2dos などのツール、テストパターンやデータ・試験結果は linux ディレクトリにしか入っていません。各々のソースディレクトリの下には、インクルードファイルを収容するディレクトリ include があります。



配布ファイルのディレクトリ構成

実証モデルの include ディレクトリにある、インクルードファイル personal_info.h には個人情報や作業環境に関する情報を集積しています。個人情報保護のため、内容の一部を削除しているので、使用する際には、皆さんの環境や情報にあわせて記入あるいは修正を行ってください。

voice ディレクトリには、音声合成用のテキストファイルだけを収納しています。お好みの声色で合成するか、自分で録音してください。シェルスクリプト create_voice にある-m オプションが声色の指定です。

```
$ ./create_voice hello
```

カオナシの画像 kaonashi.jpg、カオナシの発声 ah.wav と ahah.wav、カメラのシャッター音 camera_shutter.wav、それにテーマ曲 kaonashi-song.mp3 は、著作権保護の観点からプロジェクトファイルには含まれていません。適当なファイルを調達して、kaonashi ディレクトリ（画像）と voice ディレクトリ（音声ファイル）に収納してください。

プロジェクトファイルのダウンロードは以下から:

https://www.akiyama-tokyo.net/electronics/kaonashi.zip

コラム　ロボットのいない未来

未来の物語なのに、ロボットが全く出てこない小説もあります。代表的なのはアジモフの『銀河帝国の興亡』と田中芳樹の『銀河英雄伝説』でしょう。どちらも宇宙を舞台にしてはいるものの、ローマ帝国か中世ヨーロッパのような世界が描かれています。でも、『スター・ウォーズ』の世界観も似たようなものなのに、こちらにはさまざまな「ドロイド」が出てきますよね。先の二作は、そういった背景のもとで「人間」を描こうとしたのだと思います。もっとも、アジモフは晩年になって、「銀河帝国とロボットの統合」を目指した作品を書いていますが。

ロイス・マクマスター・ビジョルドの『ヴォルコシガン』シリーズでは、機械的なロボットがいない代わりに、バイオ技術によるクローンや（遺伝子操作による）強化人間が、うじゃうじゃと出てきます。マーサ・ウエルズの『マーダーボット』シリーズの宇宙では、人間と強化人間（一種のサイボーグ）と、「警備ユニット」が共存しています。警備ユニットとは、機械的な骨格を生体組織で包み、脳も電子回路とニューロンが結合した、一種の人造人間ですが、感情も少しはあります。（創作された）未来世界では、こういった「柔らかいロボット」と超 AI が主流になるのかもしれませんね。

Raspberry Pi 中の上級電子工作

対話型秘書ロボット

マルチプロセス・マルチプロセッサ化へのアプローチ

2020 年 8 月

著者・発行者　秋山忠次 (chuji@akiyama-tokyo.net)

非売品

参考にさせていただいたデザイン

© Walt Disney Studios

© Studio Ghibli

© Lucusfilm

© Walt Disney Studios

© 手塚プロ & 講談社

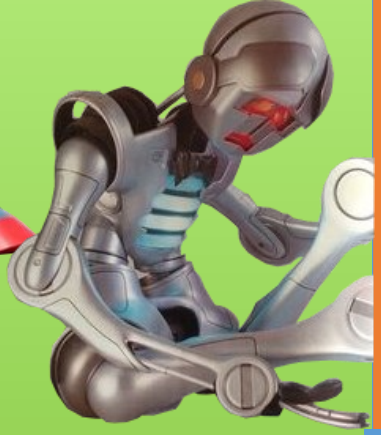
© Orion Pictures

© CBS

© 石ノ森章太郎 & フジテレビ

© 日本サンライズ & バンダイ

© 横山光輝



非売品