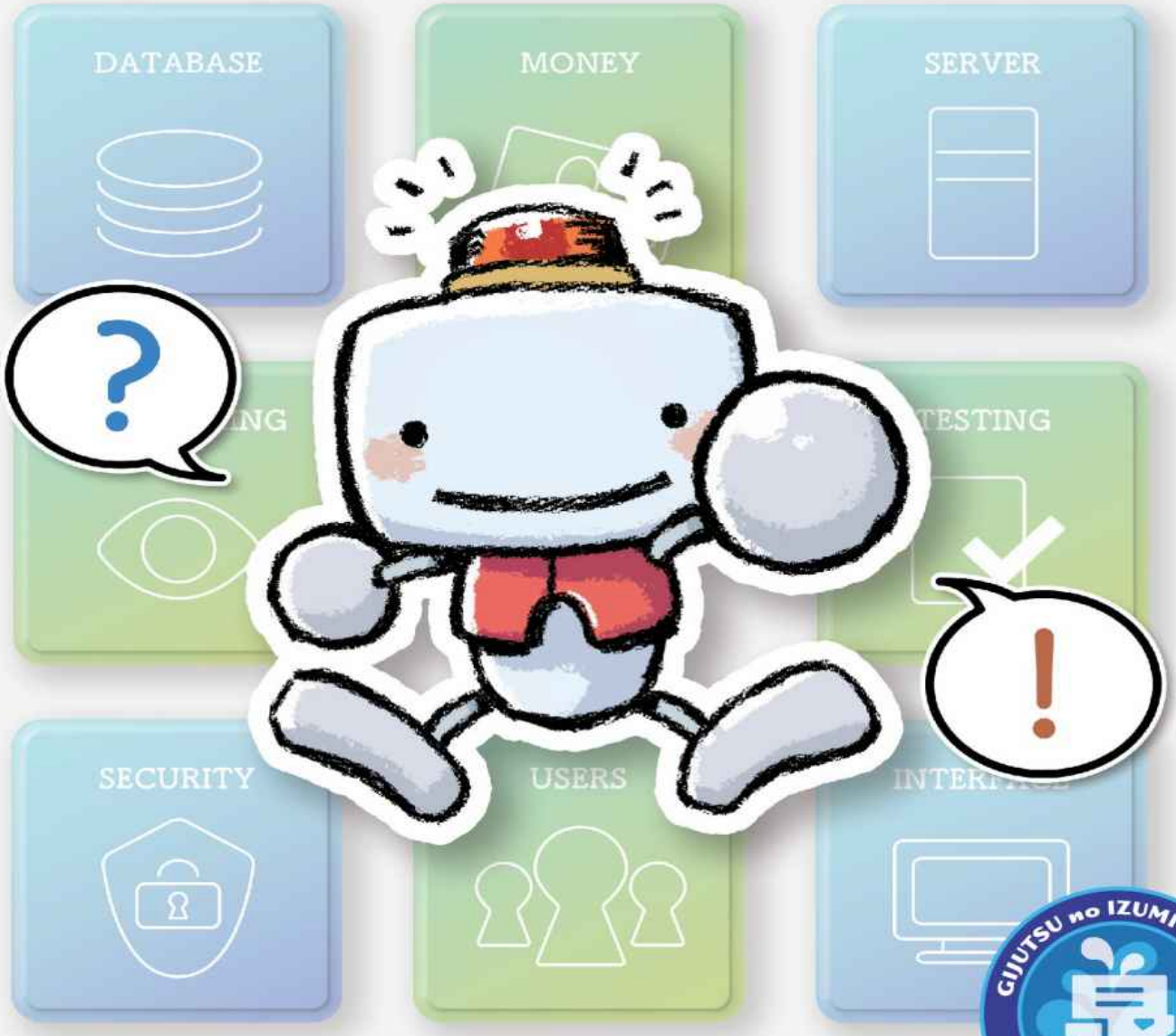
インプレスR&D［NextPublishing］

技術の泉 SERIES
E-Book / Print Book

50万人が使うDiscord Bot「shovel」の舞台裏

個人開発サービス運営実践入門

北浦 望 著



セキュリティ、稼働率、お金……
個人開発ってこんなに楽しい!?

impress R&D
An impress Group Company

GIJUTSU no IZUMI SERIES
POWERD by NEXTPUBLISHING
泉

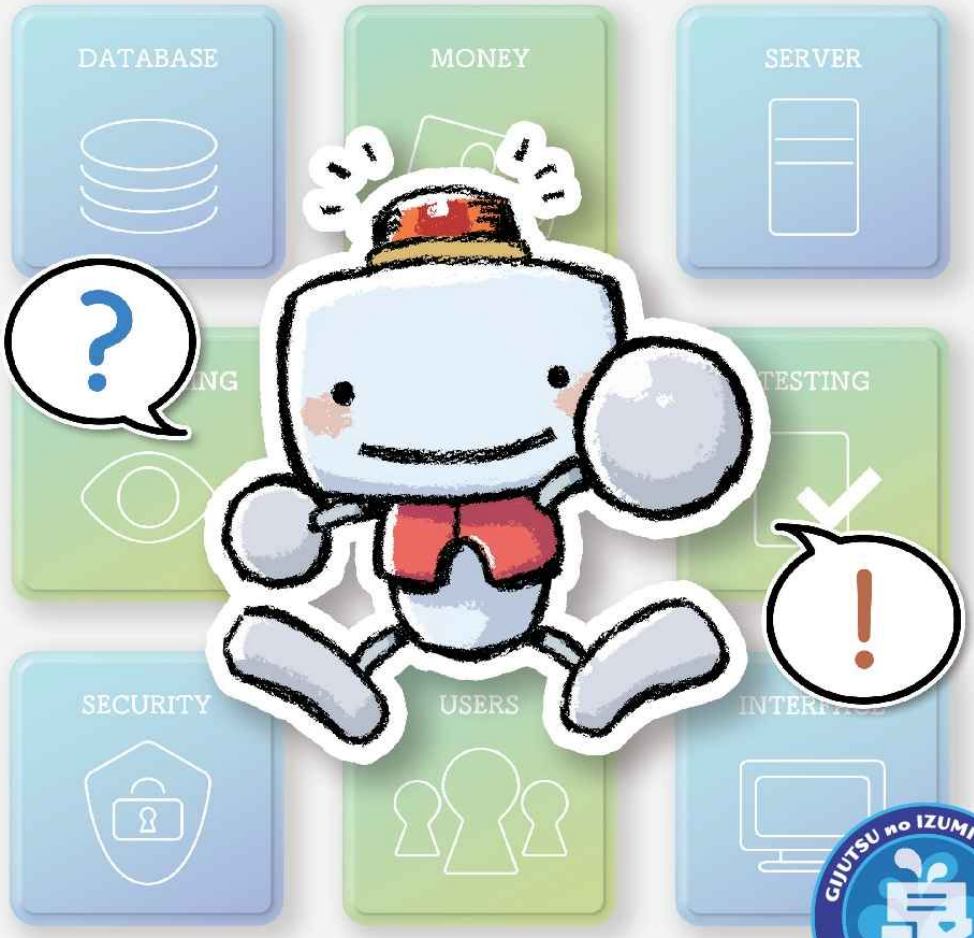
技術の泉 SERIES

インプレスR&D［NextPublishing］
技術の泉 SERIES
E-Book / Print Book

50万人が使うDiscord Bot「shovel」の舞台裏

個人開発サービス運営実践入門

北浦 望 著



セキュリティ、稼働率、お金……
個人開発ってこんなに楽しい!?

impress R&D
An impress
技術の泉

電子書籍閲覧に関するご注意

本書では、プログラムリストに専用の等幅フォントを使用しています。ビューアによって以下の作業が必要になります。

・Kindle Paperwhiteの場合：フォント設定画面で「出版者のフォント」を選択
・kobo Androidアプリの場合：フォント画面で「オリジナル」を選択

目次

はじめに

あなたのサービスをたくさんの人に届けよう！

筆者は2019年5月、日本語読み上げDiscord Bot「shovel（シャベル）」を一般公開しました。

「shovel」を一般公開すると決意したのは2018年9月。当時の筆者はBtoBソフトウェア開発経験はあれど、インフラから運営まで一手に担当する個人開発サービス運営の経験は皆無。技術面から運営面までわからないことでいっぱいでした。本当に多くの壁にぶち当たりましたが、そのたびに調査したり有識者に助言を受けたりすることで、なんとか乗り越えてきました。当初は想像もしていませんでしたが、shovelは公開から1年間で50万人のユーザーに利用してもらえるまでに成長しました。

本書はその経験を題材・実例として「サービスを作り、一般公開し、知ってもらい、運営する」うえで必要なこと（つまり、2018年9月当時の筆者が知りたかったこと）についてまとめたものです。

本文中の実例は、Discord Botで説明します。ですが、フロントエンドがDiscord Botであるというだけで、Discord以外のBotやWebサービスなど、さまざまなサービスと共通である内容も多く含まれます。もちろん、Discord Botを公開したい方にとっては、すぐに実用に供することのできる有用な情報となるはずです。

「作った！　動いた！　そこまではできたけど……」

そんな個人開発者が、もう一歩踏み出す場面で、この本が道しるべのひとつになることを願います。

2020年8月　北浦 望

対象読者

・プログラミング、サービス運営に興味がある人
・日本語読み上げDiscord Bot「shovel」に興味がある人
・Discord Botを本格的に運営したい人
・サービスを作ってみたが、公開するのになんとなく躊躇している人
・サービスを公開しているが、これでいいのか不安な人
・サービスを公開したが、思うようにユーザーが増えなかった人

この本を読むとわかること

・サービスをたくさんの人に使ってもらう方法
・サービスの土台となるサーバー設定
・サービスを攻撃から守るセキュリティ設定
・よいサービスを作るための開発プロセス
・サービス運営の障害と乗り越え方
・Discord Bot「shovel」の設計

この本の構成

第1部では筆者の個人開発サービスであるDiscord Bot「shovel」を解剖し、システム構成とコードの内部構造を説明します。Discord Botに興味がない方も、第2部以降を理解するために、軽く目を通しておいてください。第2部では、よいBotを作るための考え方と、開発プロセスについて説明します。第3部では、shovelの稼働率99.98%がどのように裏打ちされているかをお見せします。第4部ではshovelの運営とお金のことについて、個人開発で限られたリソースをどのようにやりくりしているか、工夫していることをお話します。

第1章　DiscordとDiscord Bot

　この章では、本書のさまざまな解説の例として扱うDiscord Bot「shovel」についての理解を助けるために、Discordというサービスについて簡単に説明します。また、discord.pyを使い、Pythonで簡単なDiscord Botを作ってみます。すでにDiscordやdiscord.pyを知っている方は読み飛ばしていただいてかまいません。

1.1　ゲーマーに大人気！ボイスチャットサービスDiscord

1.1.1　ゲーマー向けボイスチャットサービス

Discord（ディスコード）は、2015年に開始したゲーマー向けのボイスチャットサービスです。公式ウェブサイトによると、ユーザー数は全世界でなんと2.5億人とのことです。基本的な機能はすべて無料で利用できます。ゲーマーの間では、オンラインプレイでボイスチャットをする手段として、第一の選択肢としてあがるサービスです。ボイスチャットがメインのサービスではありますが、テキストチャットも利用できます。代替ツールとしては、Skype、LINEなどが挙げられます。

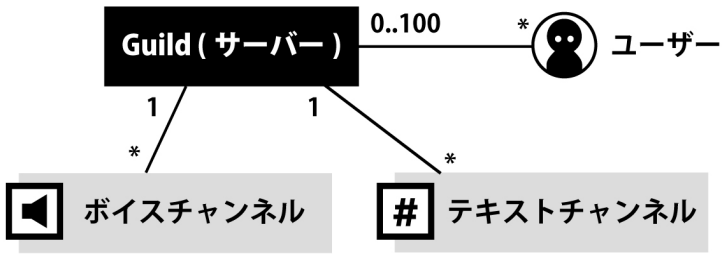
1.1.2　Discordの仕様

Discordでは多彩なコミュニティを作ることが可能です。任意のメンバーを集めたクローズドなコミュニティから、コミュニティに参加するための招待URLを公開しているオープンなコミュニティまで、Discordにはさまざまなコミュニティがあります。

Discordでは、このコミュニティを指して「サーバー」と呼びます。ユーザーが「サーバー」を作成し、そこに他のユーザーを招待します。しかし「サーバー」という名前を本書で使うと、一般名詞の「サーバー」と混同してしまいややこしいので、本書では、Discordの開発者用公式ドキュメントに倣って「Guild」と呼ぶことにします。

Discordユーザーは、100個までのGuildに参加できます。それぞれのGuildにはいくつかの「チャンネル」があります。テキストチャットは「テキストチャンネル」、ボイスチャットは「ボイスチャンネル」で行います（図1.1）。ボイスチャットは「誰かに電話をかける」というような一般的な通話とは異なり、ボイスチャンネルに入室し「誰かが参加するのを待つ」または「すでに始まっている通話に参加する」というスタイルで行います。

図1.1: Discordの仕様



1.1.3　Discord Botについて

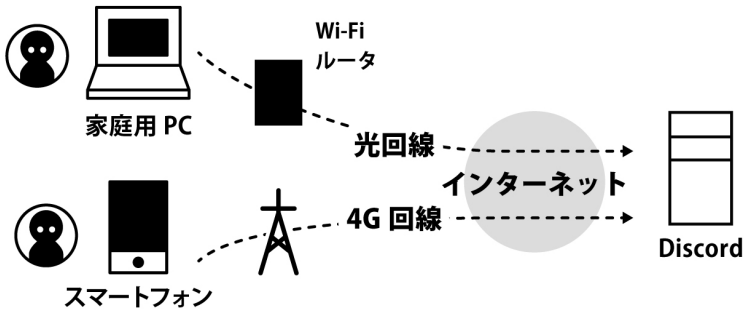
TwitterやSlackと同じように、DiscordでもBotを作ることができます。会話ログの保存をしたり、特定単語に反応するような単純なBotから、Botを通じてRPGやカードゲームがプレイできるようなBotまで、さまざまなBotが存在しています。つづいて、Botの仕組みについて詳しく解説します。

1.2　Discord Botの正体

1.2.1　Botはどこで動いているか

そもそもBotとは何なのでしょうか。まず、Botではない通常のユーザー（人間）がDiscordを利用する場合の接続方法について見ていきましょう（図1.2）。ユーザーは、光回線や4G回線でインターネットに接続された端末（スマートフォン、PC）を持っています。この端末から、アプリやWebブラウザを通じてDiscordにアクセスしています。

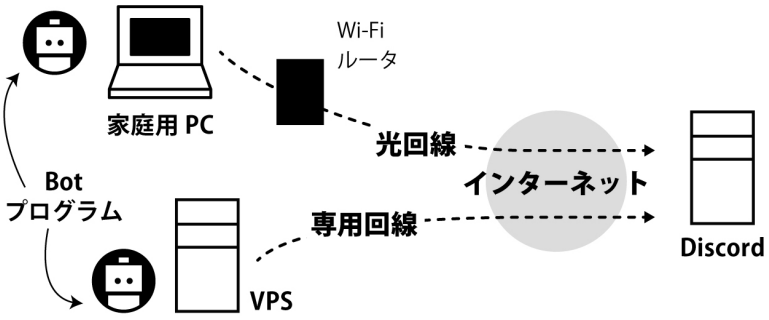
図1.2: Discordのアプリから接続する通常ユーザー



では、Botはどうでしょう。人間が操作するのではなく、プログラムだけがDiscordに接続するのがBotです（図1.3）。Botには普段見慣れたDiscordアプリのような操作画面は通常存在しません。黒い画面でコマンドをタイプして動かすプログラムが、Botを操作します。

たとえば誰かのテキストチャット発言をキャッチし、それにテキストで反応するようなプログラムを書くと、Discordのテキストチャットで反応するBotができあがります。

図1.3: Discordに接続するBotプログラム



1.2.2　Discord Botアカウントとは

Discord Botを動かすには、まずDiscordのWebサイトでBotアカウントを作成します。すると、トークンという文字列が割り振られます。Botのプログラムはそのトークンを APIサーバーに渡すことで、Botとしてログインできます。トークンは、いわばIDとパスワードがセットになったようなものです。

1.2.3　Botが稼働するために必要なもの

上記の説明のとおり、Botは最低限「プログラムを動かすコンピューター」と「インターネット回線」と「トークン」があれば、家庭用のPCでもどこでも動かすことができます。次の節では、あなたのPCで実際にBotを動かしてみましょう！

1.3　Discord Botで「Hello, world!」

1.3.1　discord.pyとdiscord.ext.commandsフレームワークの紹介

discord.pyは、PythonでDiscord Botを作る際にもっともよく使われているAPIラッパーです。特徴として、モダンなPythonインターフェイス、レートリミット制御、Discord API完全準拠、速度とメモリ効率の両立が挙げられます。GitHubでソースコードが公開されています。

discord.pyには、discord.ext.commandsという拡張フレームワークがあります。拡張フレームワークというと敷居が高い印象を受けますが、このフレームワークを使うとBotが簡単に実装できます。筆者が開発しているshovelも、このフレームワークに沿って設計されています。

ここではdiscord.ext.commandsフレームワークについて学びながら、小さなBotを作ってみましょう。

1.3.2　前提条件

discord.pyを動かすには、Python3.5.3以上が必要です。お手元の環境に3.5.3以上のPythonが使える環境をご用意ください。また、「Botアカウント」・「トークン」を事前に作成・取得しておいてください。作成したBotアカウントは、自分の管理するGuildに追加しておきます。Botアカウント・トークンについては、筆者の記事（https://cod-sushi.com/discord-py-token/）を参考にしてください。

1.3.2.1　CLIを開く

本節でのライブラリーのインストールやプログラムの実行は、OS上のCLIから行います。Linuxならターミナル、Windowsならコマンドラインプロンプトを開いてください。Windowsでコマンドラインプロンプトを開くには、windowsメニューを開いてcmdと入力し、エンターキーを押します。

1.3.2.2　discord.pyをインストールする

Bot開発を支援するライブラリー「discord.py」をインストールしましょう。リスト1.1のOSコマンドを実行します。

リスト1.1: discord.py インストール方法

```
$ python3 -m pip install -U discord.py<Enter> ←Linuxの場合
$ py -3 -m pip install -U discord.py<Enter> ←Windowsの場合
```

次に、正しくインストールが完了したか確認します。リスト1.2のOSコマンドを実行してください。

リスト1.2: discord.pyがインストールできたか確認する

```
$ python3 -m pip show discord.py<Enter> ←Linuxの場合
$ py -3 -m pip show discord.py<Enter> ←Windowsの場合
```

数行のパッケージ情報が出力されれば、正常にインストールできています。

1.3.3　BotをDiscordに接続してみよう

手始めに、BotをDiscordに接続するコードを書きます。テキストエディターを開き、リスト1.3の内容を入力してください。入力できたら、「bot.py」という名前を付けて保存します。

リスト1.3: BotをDiscordに接続する（bot.py）

```
from discord.ext import commands


TOKEN = "<取得したトークン>"
bot = commands.Bot("?")


bot.run(TOKEN)
```

この簡単なコードを実行してみましょう。CLIにリスト1.4のように入力します。

リスト1.4: Botプログラム実行方法

```
$ python3 bot.py<Enter> ←Linuxの場合
$ py -3 bot.py<Enter> ←Windowsの場合
```

このOSコマンドを実行したら、確認のため一度Discordを開きます。あなたのGuildにいるBotがオンライン（緑色の表示）になっていたら成功です！

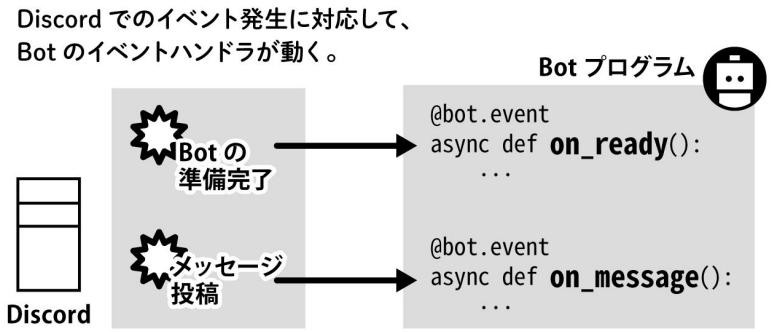
1.3.4　イベントを使ってみよう

さて、BotとしてDiscordにログインする処理は実装できました。ですがこのままでは、このBotは何の役にも立ちません。そこで、イベントに反応して処理をするようコードを変更してみましょう。イベントとは、Discord上で発生するさまざまな出来事です。イベントの例の一部を以下に挙げます。

・Botが準備完了したとき（on_ready）
・チャンネルにメッセージが投稿されたとき（on_message）
・Guildからメンバーがいなくなったとき（on_member_remove）

その他すべてのイベントについての情報は、後述するdiscord.pyの公式ドキュメントに掲載されています。それぞれのイベントについて、イベントごとに呼ばれるイベントリスナーのメソッド名が決まっているので、そのメソッドで処理内容を定義します（図1.4）。

図1.4: イベントを起点とした動作



では、「Botが準備完了したとき」のイベントを拾って、ログインしたBotアカウントのユーザー名をCLI上に表示する処理を実装してみましょう。リスト1.3で書いたコードのbot.run(TOKEN)の直前に、リスト1.5の内容を追加してください。

リスト1.5: イベントを使った処理

```
@bot.event
async def on_ready():
    print(f"ログイン完了 ユーザー名: {bot.user.name}")
```

動作確認を行います。先ほどと同じく、リスト1.4のようにOSコマンドを実行してください。CLIにログイン完了 ユーザー名: <Botアカウントのユーザー名>と出力されたら成功です。

1.3.5 Botのコマンドを定義してみよう

さて、いよいよ最後の仕上げです。ユーザーからの「コマンド」を受け取って処理を行うようコード変更してみましょう。この「コマンド」とは、ここまでに出てきた「OSコマンド」とは違うものです。本書では、単に「コマンド」と言った場合Botのコマンドを指し、「OSコマンド」と言った場合はCLIで実行可能なコマンドを指します。

Botにコマンドを実装したいときは、コマンド名をそのままメソッド名に採用し、@bot.command()デコレータを付けたメソッドを定義すれば完了です。リスト1.6をごらんください。このコードを、先ほどと同じく、

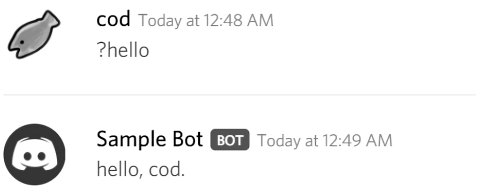
bot.run(TOKEN)の前に追加します。

リスト1.6: コマンドの定義

```
@bot.command()
async def hello(ctx):
    await ctx.send(f"hello, {ctx.author.name}.")
```

最後の動作確認です。Discordを開いて、?helloと入力してみましょう。うまくいけば、図1.5のように、あいさつをしてくれるはずです。

図1.5: コマンドによるユーザーへのあいさつ



1.3.6　おつかれさまでした！

これでBotがひとつ完成しました！　ネットワークプログラミングやDiscord APIへの接続のプロトコルの知識もまったく必要なく、簡単に書けるのがおわかりかと思います。DiscordでBotを作りたければ、このフレームワークを利用して開発すると効率が良いでしょう。最後に、作ったBotの全体像をリスト1.7に掲載しておきます。

リスト1.7: Botのコード全体（bot.py 完成版）

```
from discord.ext import commands


TOKEN = "<取得したトークン>>"
bot = commands.Bot("?")



@bot.command()
async def hello(ctx):
    await ctx.send(f"hello, {ctx.author.name}.")
```

```
@bot.event
async def on_ready():
    print(f"ログイン完了 ユーザー名: {bot.user.name}")


bot.run(TOKEN)
```

1.3.7　実用Bot実装のヒント

上記では本当に最小限のコードのみ示しましたが、実際にBotを作る場合は、もっと多様な処理を入れたくなるはずです。そのときに参考となるのは、「discord.py公式ドキュメント」と「RoboDanny」です。

1.3.7.1　公式ドキュメントを読もう

discord.pyにできることは、すべて公式ドキュメントに書いてあります。公式ドキュメントは英語で書かれていますが、基本的な機能については有志によって日本語に翻訳されています。ドキュメントは、discord.pyのGitHub（https://github.com/Rapptz/discord.py）からリンクされています。

1.3.7.2　RoboDannyを参考にしよう

Botを開発するうえで、discord.py開発リーダーであるDanny氏が公開しているRoboDannyというBotのコード（https://github.com/Rapptz/RoboDanny）は非常に参考になります。shovelのクラス構成も、RoboDannyを一部参考にしています。RoboDannyは、discord.pyのサンプルコードの宝庫であり、discord.pyのベストプラクティスです。実装や設計に迷ったときは、一度RoboDannyのコードを読むことを推奨します。

第2章　shovel - 日本語読み上げDiscord Bot

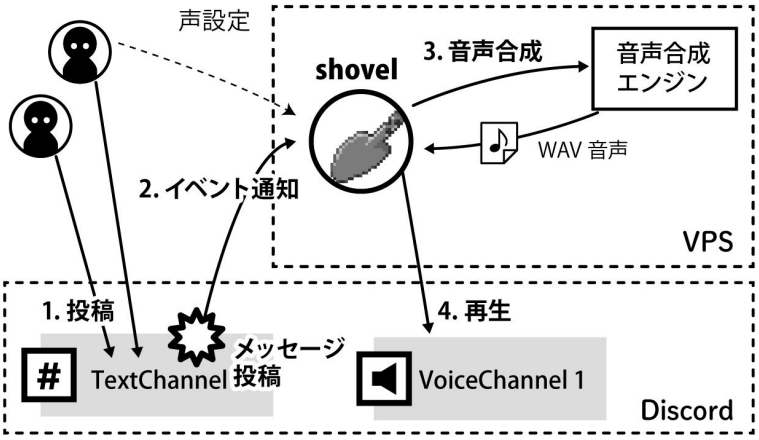
　この章では、筆者が運営する個人開発サービスである日本語読み上げDiscord Bot「shovel」について紹介します。shovelが開発された経緯、shovelの機能、ユーザーの声など、個人開発サービスの開発・運営の実例としてお読みください。

2.1　50万人が使う読み上げBot「shovel」

2.1.1　shovelとは

shovelは、Discordのテキストチャットの内容を音声合成し、ボイスチャンネルに流すBotです（図2.1）。このような機能は、TTS（Text To Speech, 読み上げ）と呼ばれます。発言ユーザーごとに違う声で読み上げるので、慣れると音声を聞いただけで発言者が誰なのかわかる、というのが大きな特徴です。

図2.1: shovelの機能



shovelを使ってみたい方は、まずはshovelの公式Twitterアカウント（https://twitter.com/shovel_discord）をごらんください。

2.1.2　開発の動機

shovelを開発するきっかけとなったのは、筆者のゲーム仲間でした。2018年初旬、Twitterを経由して親しくなった4人でDiscordのGuildに集まり、ボイスチャットを繋いでゲームをすることが筆者の日課になっていました。筆者はマイクを使って話していましたが、集まるのが主に深夜帯であったこともあり、友人のうちふたりはボイスチャットを聞くことはできても声を発することは難しい、いわゆる「聞き専」でした。会話だけをしているときはDiscordの画面を見ることができるため不便ではないのですが、ゲームに集中していると聞き専の人の発言を見逃してしまいます。見逃しが申し訳ないと筆者が発言したところ、ゲーム仲間のひとりが「チャットを読み上げてくれるBotがあるといいのではないか」と言いました。これがshovel誕生のきっかけです。

2.1.3　開発の経緯

2018年3月、shovelの前身である「しゃべりマス」が誕生しました。当時は、前述のGuildや、筆者のごく親しい友人の管理するGuildのみで利用していました。このときは複数Guildでの利用は想定しておらず、異常系への対応も非常に甘いものでした。また、筆者の家庭用PC（Windows OS）上の仮想環境で動いているUbuntuで稼働させていたため、Botは普段オフラインになっており、使用したいときにはPCおよび仮想環境を立ち上げる必要がありました。ただしユーザーごとに声を設定できるという機能はこのときすでに実装されていました。

実のところ、筆者は周囲の親しい友人に使ってもらったことで満足していました。しかし、前述のゲーム仲間から強い後押しをもらい、誰かの役に立てるのであればと世の中へ公開することに決めました。2018年9月、「talker」という開発名を付け、実用化に向けたコード修正を開始しました。shovelの原型はこのコード修正でできあがりました。正確には修正というより、ほぼすべて作り直したといえます。複数Guildでの利用を想定した作りに変更するなどの機能面を強化したほか、親しい友人向けだったために緩い言葉遣いだったメッセージを修正する作業もこのときに行いました。

そして、令和はじめの日である2019年5月1日、ついに「shovel」という名称でサービス開始したのです。2020年8月現在も、継続的に利用Guildは増加しつづけています（図2.2）。

図2.2: shovel開発の経緯

2.1.4　基本情報

2020年8月現在、ユーザー数は50万人超、導入Guildは約4万となっています。ピークタイムには、同時に1500程のGuildで、1分間あたり1万文字以上の読み上げを行っています。音声の合成および再生を行うという性質上、必要とされるスペックは大きいです。

shovelは、KAGOYA JapanのVPSで動作しています。shovelの稼働率はスリーナインクラス（99.98%）で、30日あたりの停止時間は5分以下です。

2.2　shovelの機能

ここでは、shovelの機能について簡単に説明します。すべてに目を通す必要はありませんが、今後の解説でshovelを例に説明する場面がありますので、都度ご確認ください。また、shovelのTwitterアカウントで招待URLを紹介しています。本書を読んでいただく際には、shovelの動きを思い浮かべると理解もスムーズです。Discordをお使いの方は、ぜひ一度触ってみてください。

2.2.1　基本的な使い方

shovelは、!shではじまる文字列をテキストチャンネルに送信することで操作します。このような文字列を「コマンド」と呼びます。

2.2.1.1　読み上げの実施

!sh sコマンドでそのチャンネルを読み上げを開始します。メンバーが全員ボイスチャンネルから抜けるか、!sh eコマンドが呼ばれると、読み上げを終了します。

2.2.1.2　単語登録

!sh awコマンドで、単語と読みを辞書に登録できます。登録した単語は、そのGuildでの読み上げにのみ使用されます。!sh export_wordで辞書のファイルへのエクスポート、!sh import_wordでファイルからの辞書インポートが行えます。

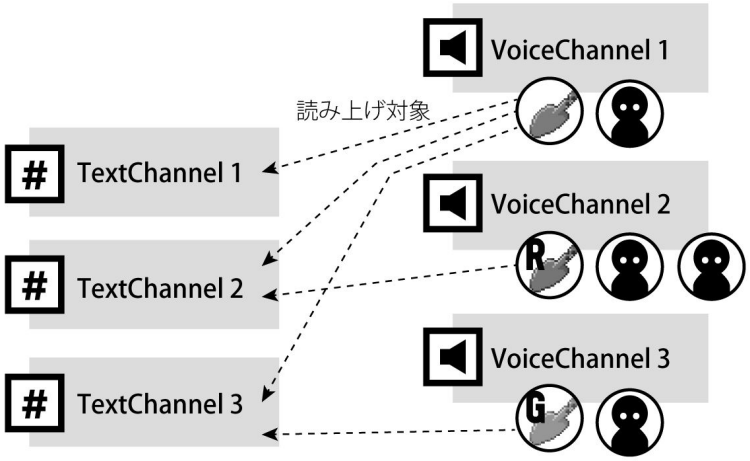
2.2.1.3　その他の機能

shovelには、他にもさまざまな便利な機能があります。詳細は前述のTwitterアカウントで紹介している説明書をごらんください。

2.2.2　shovel RGB

Discord Botは、ひとつのGuildにつき同時にひとつのボイスチャンネルにしか接続できません。shovelでは、同時に複数のボイスチャンネルに接続して読み上げができるよう、3機体制を取っています。青いアイコンのメインshovel「shovel」と、赤・緑アイコンのふたつのサブshovel「shovel_red」「shovel_green」があります。サブshovelに対してはGuild設定や声設定は行えず、読み上げの開始・停止コマンドのみ使えます。赤、緑、青と3色揃っているので、これらを「shovel RGB」と呼んでいます（図2.3）。

図2.3: shovel RGB


VoiceChannel 1
読み上げ対象
TextChannel 1
VoiceChannel 2
R
TextChannel 2
VoiceChannel 3
G
TextChannel 3

2.3　shovelユーザーの声

2.3.1　アンケートの概要

2019年9月6日から8日にかけて、shovelユーザーにアンケートを実施し、100件以上の回答を頂きました。ここでは、そのアンケートの結果をもとに、shovelがどのように使われているかをご紹介します。

2.3.2　アンケート結果

2.3.2.1　shovelの用途

まずは、shovelの用途についてです。図2.4をごらんください。Discordのユーザー層のとおり、圧倒的にゲーム中の通話での利用が多いようです。次いで雑談、作業通話（絵を書くなどの作業をしながら通話すること）となりました。「その他」では、用途を問わず、聞き専の方がいるときに使っているとのことでした。

図2.4: shovelの用途 アンケート結果

2.3.2.2　shovelの好きなところ

次に、shovelの好きなところを聞きました。自由入力で回答していただき、分類して集計した結果が以下です。これ以外にも、その他、「素直なところ」「アイコンがかわいい」「全部」などの回答をいただきました。

・ユーザーごとに細かく声を設定できる（33件）
・辞書登録によってカスタマイズできる（10件）
・画面を見なくてもテキストチャットの内容がわかる（9件）
・声を出せない人とも一緒に楽しく会話できる（8件）
・話し方がかわいい、聞きやすい（7件）
・遅延が少ない、動作が軽い、高品質（6件）
・自分好みにカスタマイズできる（5件）

・運営者の対応速度、要望の反映（3件）

・複数のボイスチャンネルで使える（2件）

声設定機能は「しゃべりマス」開発初期からあったshovelのコアと言える機能であり、ユーザーもこの機能を便利に感じていることがわかります。また、「画面を見なくてもテキストチャットの内容がわかる」「声を出せない人とも一緒に楽しく会話できる」という点は、まさにshovelを使って実現したいと筆者が常日頃考えていることであり、ユーザーがDiscordを楽しむ手伝いができていると受け止めています。また、Botとしての品質の高さ、運営を評価するコメントも頂いています。

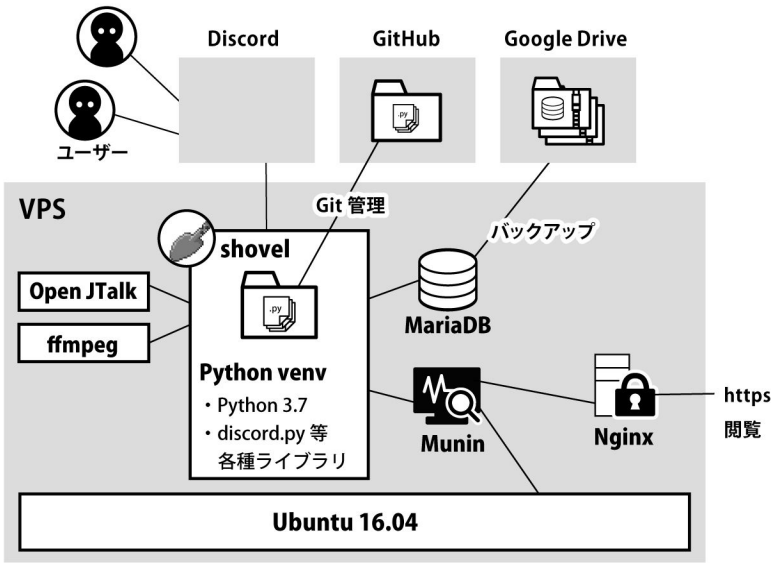
第3章　shovelのシステム構成

　前章では個人開発サービスの表側について述べました。つづいて、個人開発サービスの舞台裏を覗いてみましょう。この章では、shovelの全体構成について図を使って説明し、組み込んだプロダクトの概要とそのスペックについて紹介します。それぞれの構成要素についての詳しい説明や採用理由は第12章「Botの土台作りのダイジェスト」をごらんください。

3.1　shovelのシステム全体構成

　shovelはVPS上で稼働しています。shovelのシステム全体構成を図3.1に示します。VPSには、shovel本体のコードだけでなく、OS、ミドルウェア、外部アプリケーション、外部パッケージが入っており、それらが協調して動作することによりshovelのサービスとして稼働しています。

図3.1: shovelシステム全体構成



　この章では、OSを含むサーバースペック、ミドルウェアと監視用のツールを紹介します。第4章「shovelのソフトウェア構成」では、shovel本体のコードや、外部アプリケーション、外部パッケージについて説明します。

3.2　shovelの必要スペックは？　サーバー性能公開

3.2.1　家庭用PCからVPSへ

第2章でも述べましたが、shovelの前身であるBot「しゃべりマス」は家庭用PCで稼働していました。親しい友人だけで使っていましたので、負荷は少なく、信頼性が低くても問題にならないため、わざわざサーバーを借りる必要がなかったからです。しかし、サービスとして公開するにあたり、図1.3で示したようにVPSで稼働させることにしました。

3.2.2　VPS

2020年8月現在、KAGOYA JapanのKVMプランを利用しています。他社と比較して性能に対しての価格が安いという理由で契約しましたが、サポート品質も他社サービスに引けを取りません。これまでに数回問い合わせなどを行いましたが、すべてのケースで迅速で丁寧なサポートのサービスを受けられ、大変満足しています。

3.2.3　サーバースペック

OSはUbuntu16.04、メモリ8GB、CPUは6コアです。2020年8月現在、shovelのピークタイムにおいて、メモリ・CPUともに80%ほど使用しています。

3.3　shovelを助けるミドルウェアたち

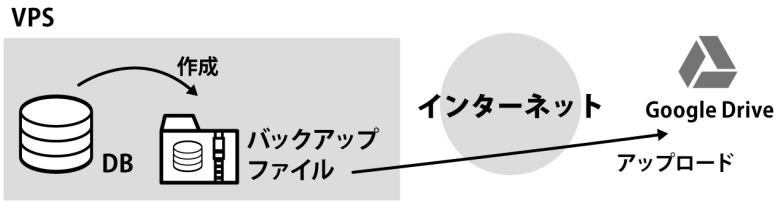
3.3.1　DB

shovelでは、データを永続化する手段としてDBを採用しています。MySQLから派生したオープンソースのDBMSであるMariaDBを採用しています。DBサーバーはshovelと同じVPS上に構築してあります。

3.3.1.1　DBのバックアップ

データが不慮の事故で消失してしまわないように、バックアップをとっています。しかし、作成したバックアップをVPS内に保存していては、VPSにアクセスできなくなるようなトラブルがあったときに意味がありません。そこで、VPSとは違う場所、具体的にはGoogle Driveに転送しています（図3.2）。

図3.2: shovelのDBデータバックアップ

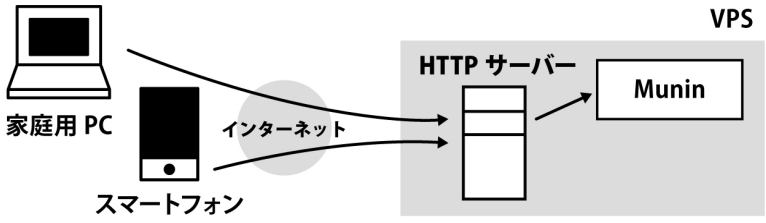


バックアップは、cronで自動化しています。毎日、接続数がもっとも少なくなる時間帯にバックアップを実施します。同様に、週次バックアップも実施しています。

3.3.2　HTTPサーバー

後述の監視ツールやログ閲覧ツールのフロントエンド用ミドルウェアを動作させるため、HTTPサーバーを設置してあります（図3.3）。HTTPサーバーとしてはnginxを採用しています。HTTPサーバーは運営のみを目的として使用していますが、Let's EncryptのSSL証明書でHTTPS化しています。

図3.3: HTTPサーバーの用途


VPS
HTTP サーバー
Munin
家庭用 PC
インターネット
スマートフォン

3.4　shovelを見守る監視体制を実現するツール

shovelでは、信頼性を高めるためにシステムの監視を実施しています。既存のツールを用いることで効率化していますので、ここではそのツールについてご紹介します。監視の目的や詳しい内容については第14章「監視 - 24時間みまもり体制」で述べていますのでご参照ください。

3.4.1　監視対象

CPU、メモリ、ネットワーク、ファイルI/Oの状況など、システムの状態を監視しています。

また、shovelの状態も監視しています。監視している指標は、読み上げメッセージ数・読み上げ文字数、メッセージの投稿から読み上げまでにどのくらいラグ（遅れ）があるか、導入Guildの数、エラー発生数など、多岐にわたります。とにかく多様なデータを取っておき、何かあったときに遡って調査できるようにしています。

3.4.2　監視ツール

監視ツールとしてMuninを採用しています。Muninとは記録されたシステムデータをWebで閲覧したり、システムデータの推移によってサーバー管理者にアラートを上げることが可能なフロントエンドツールです。Muninは、プラグインを作成することで独自の監視項目を追加できます。この仕組みを使い、shovelの各種データをMuninで収集しています。現時点の数値だけでなく継続的なデータの推移を追跡することで、今後必要となってくるスペックを見積もる際などに役立ちます。Muninの代替ツールとしてMackerelという高機能な監視ツールがありますが、無料利用の範囲では監視項目数が制限されていたため、コストの点でMuninに軍配が上がりました。

プラグインからshovelの状態を監視するために、shovelには監視ツールに状態を配信する機能が備わっています。これについては、第8章「実装 - いざ、コーディング」で述べます。

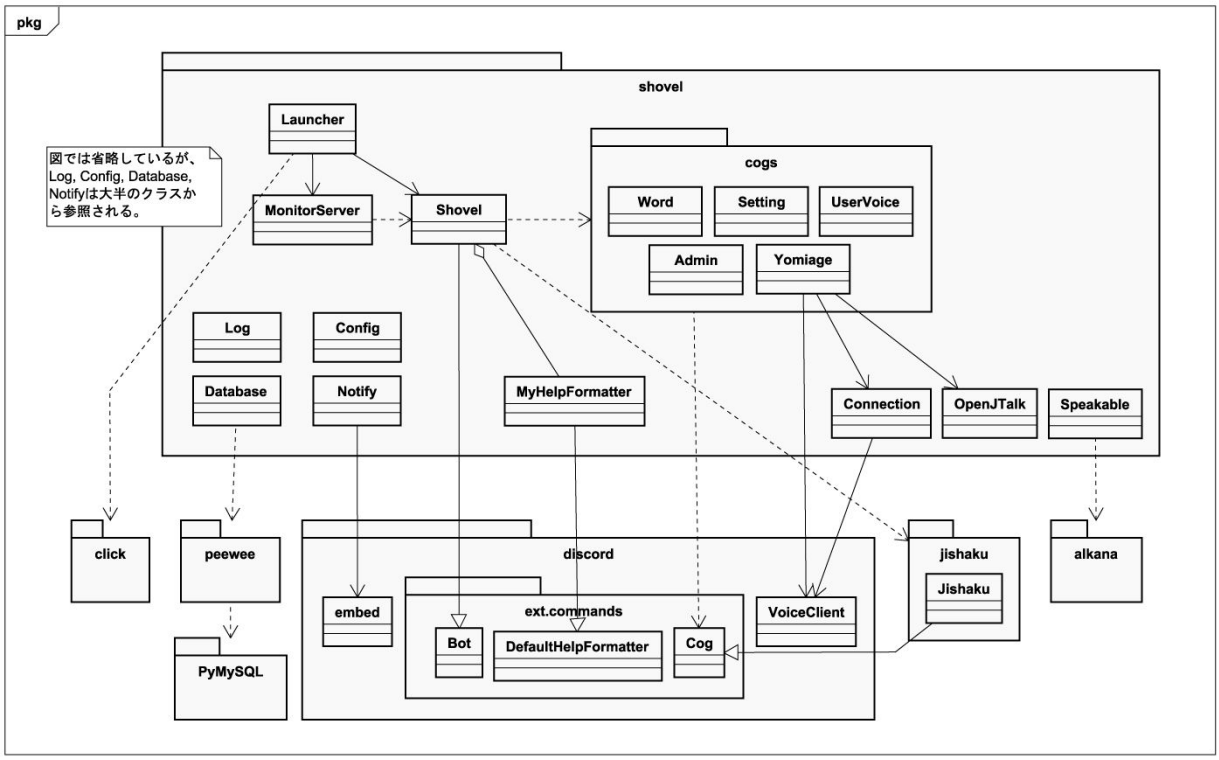
第4章　shovelのソフトウェア構成

この章では、shovelのソフトウェア構成に注目し、より詳細にサービスの内側を見ていきます。まずshovelのクラス図と読み上げ処理を題材としたシーケンス図により、shovelのクラス同士が協調して動作していることを示します。そのあと、shovelを構成する個々のパーツについて詳細に見ていきます。Pythonやdiscord.pyによる詳細な実装についても踏み込みますが、本書ではサービスの設計・実装についてはshovelを例として解説するため、Pythonやdiscord.pyによる開発をしない方も軽く目を通していただければと思います。

4.1　shovel本体を俯瞰する

4.1.1　クラス構成

図4.1: shovelクラス図



shovel本体は、さまざまなパッケージから構成されています。UML2.0に基づくクラス図を図4.1に示します。また、Open JTalk、ffmpegといった外部アプリケーションも動作に必須となります。このクラス図では、厳密にはクラスではないモジュールも便宜上クラスとして描画しています。

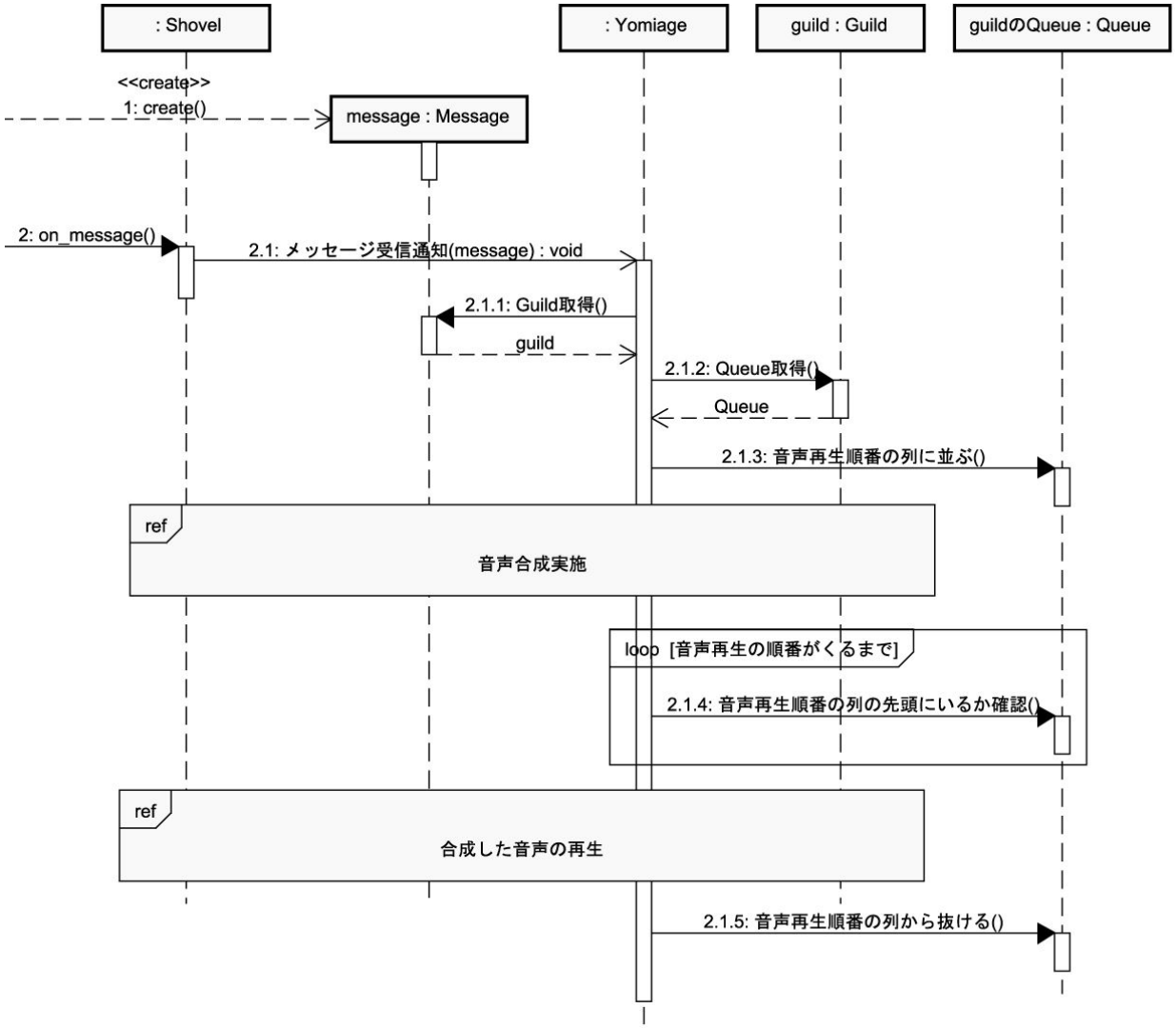
4.1.2　動作の例 - 読み上げ処理のシーケンス図

ここでは、shovelのクラス・モジュールが協調して処理をしていることの例として、読み上げ処理のシーケンス図を掲載します。

4.1.2.1　読み上げ処理の全体

読み上げ処理の全体を図4.2に示します。shovelはメッセージ受信イベントを検知後、まずは音声合成を行います。次に、合成した音声を再生します。

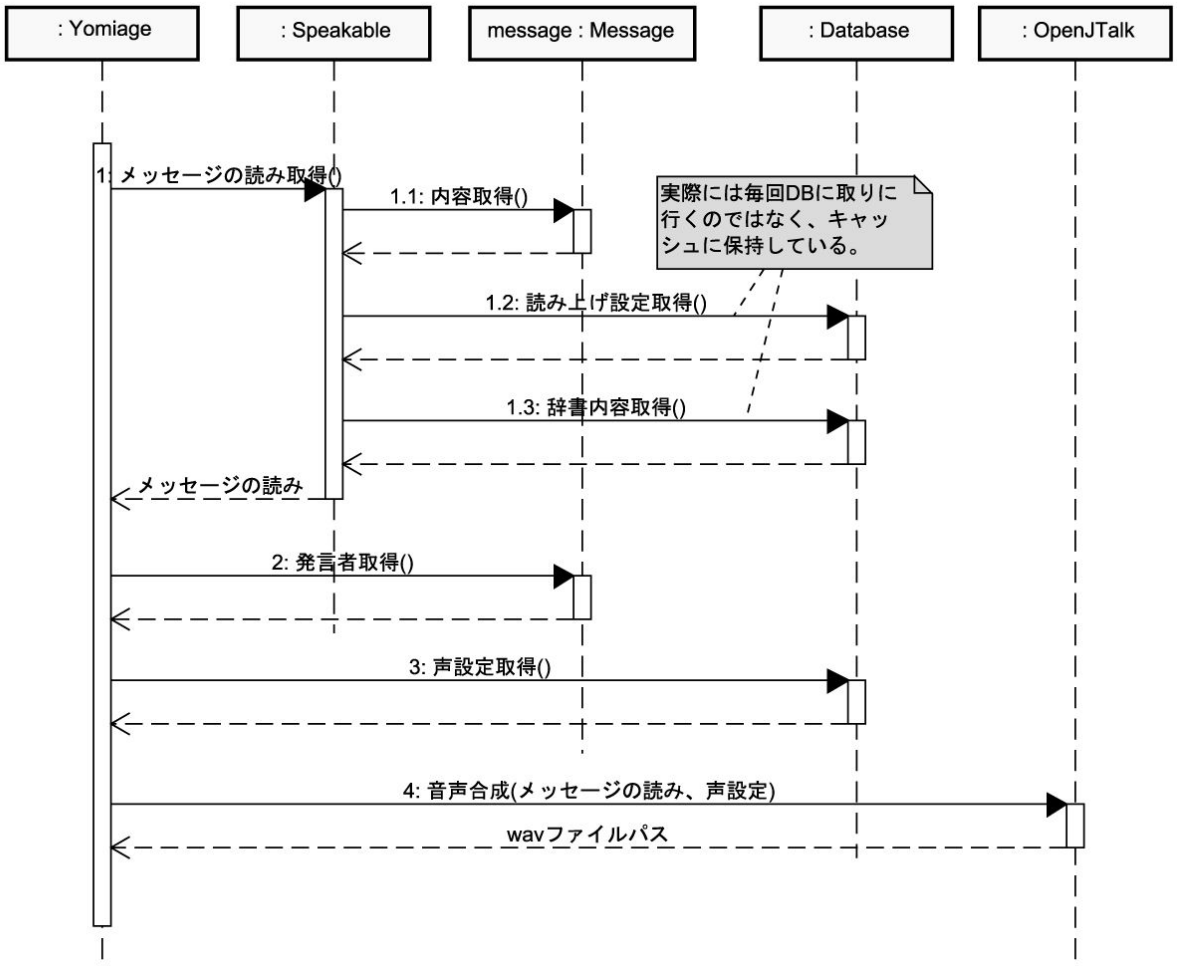
図4.2: 読み上げ処理の全体シーケンス図



4.1.2.2 音声合成

図4.3は、受信したメッセージ文字列がwavファイルになるまでの流れを示したものです。読み上げに必要なデータをDBから集め、モジュールOpenJTalkに渡すための文字列を作った後、読み上げ音声合成を行います。

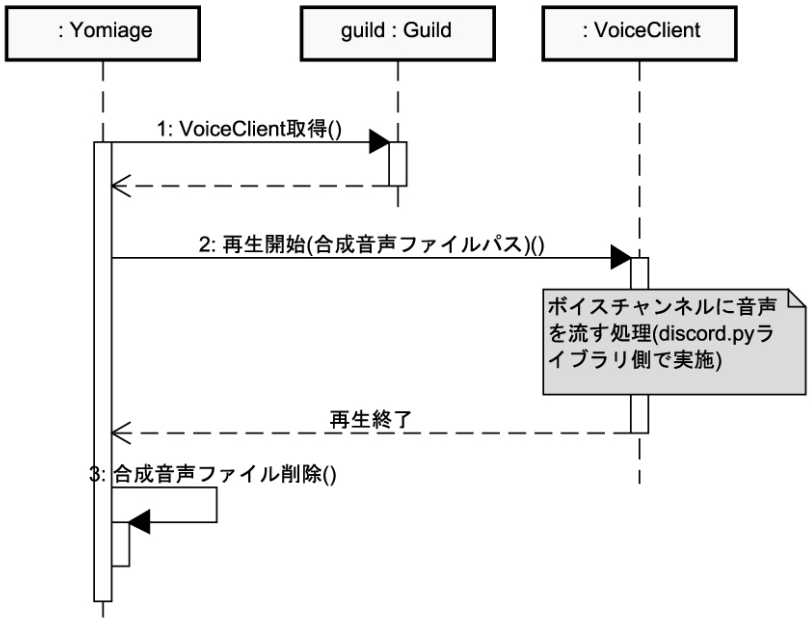
図4.3: 音声合成処理のシーケンス図



4.1.2.3　音声再生

最後に、Guildのボイスチャンネルに音声を流します（図4.4）。discord.pyのVoiceClientクラスに音声再生を指示し、再生が終了したら合成音声の入ったwavファイルを削除します。

図4.4: 音声再生処理のシーケンス図


: Yomiage
guild : Guild
: VoiceClient
1: VoiceClient取得()
2: 再生開始(合成音声ファイルパス)()
ボイスチャンネルに音声を流す処理(discord.pyライブラリ側で実施)
再生終了
3: 合成音声ファイル削除()

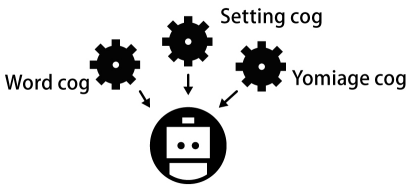
4.2　cog - shovelを構成する歯車たち

4.2.1　cogとは

図4.1にあるとおり、shovelには、cogsというパッケージがあります。このパッケージには、discord.pyのフレームワークの構成要素であるcogと呼ばれるクラスが含まれます。

cogは一言でいうと、「ある機能に関連するコマンド定義とイベントハンドラーをまとめたもの」です。このcogをBotに追加すると、さまざまな機能をもつBotができあがるというわけです（図4.5）。

図4.5: cogを集めてBotを作る



cogは英語で「歯車」という意味で、その意味が示すとおり、cogを組み合わせるとBotができあがります。cogを使わずにBotを作ることももちろん可能です。しかし、Botの機能が増えてくると、コマンドや必要となるイベントリスナーも増加します。cogを使わず、単一のファイルにBotの処理をまとめて書いていると、そのファイルがどんどん肥大します。ある機能の実現のためにコードを変更したときに、まったく関係ない機能に影響が出てしまうこともありえます。独立したcogを作って組み合わせる構成を採用すると、そのような問題が解決できます。

たとえばshovelであれば、Bot開発者である筆者が操作するためのコマンドを管理するAdmin、Guildの設定を管理するためのSetting、ユーザーの声設定を生成・保存するUserSettingなどのcogが存在します。これらのcogは基本的に相互に依存することはなく、あるcogを削除しても、他のcogに影響することはありません。

以降では、shovelのcogの一部を紹介します。

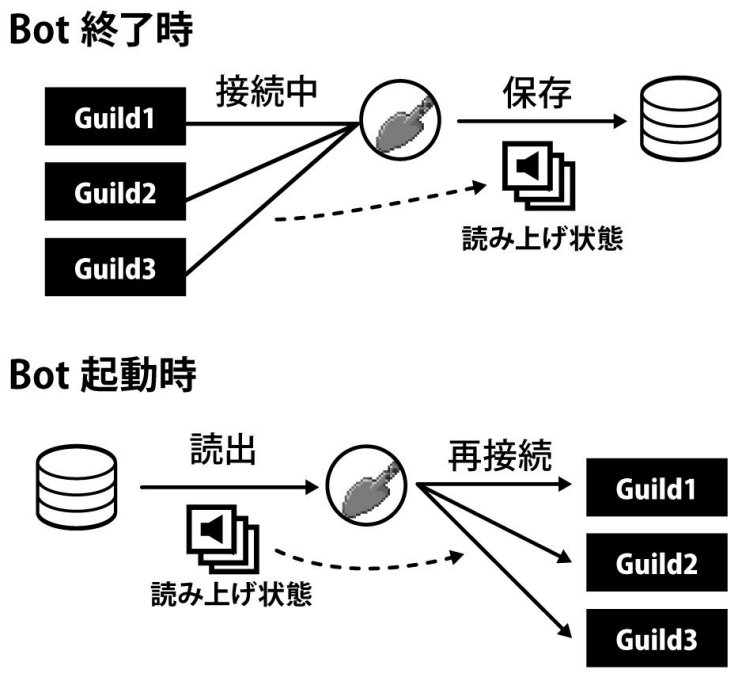
4.2.2　Yomiage

shovelのコア部分となるcogです。読み上げ機能全般について、ユーザーとのやりとりおよび各機能の連携を行います。ユーザーからの読み上げ開始指示を受けてボイスチャンネルに接続を行う、ボイスチャンネルからshovel以外がいなくなったら自動的に読み上げを終了する、読み上げ中のテキストチャンネルに投稿があ

ったら音声合成を行うなど、適切なタイミングで処理を請け負うモジュールに指令を出す役割といえるでしょう。

またこのcogはBotプログラムの停止をトリガーとして、その時点で読み上げ実施中のテキストチャンネルがあれば、Guild IDとボイスチャンネルID、読み上げ中のテキストチャンネルIDをDBに書き出します。そしてBotの開始時にはそのデータを読み出し、自動的に再接続を行います（図4.6）。これにより第6章「品質を上げるための設計のポイント」で紹介している自動復帰機能が実現できます。

図4.6: 自動再接続のしくみ



このcogは他のcogと比較してやることが多く複雑なので、実際の仕事の多くはConnection、OpenJTalk、Speakableという3つのモジュールにまかせています。これらのモジュールの責務については、次節で紹介します。

4.2.3 Admin

Bot開発者、すなわち筆者の使うコマンドが含まれます。たとえば、stop_botは、その時点で読み上げを実施しているテキストチャンネルにメンテナンスを開始する旨を通知し、すべてのボイス接続を切断してからBotを安全に停止するコマンドです。また、monitorは、Botやサーバーの負荷状況を表示するコマンドです。

このcogに属するすべてのコマンドは、discord.ext.commands.Cogクラスに備わっている、cog_checkというcog全体の権限を設定する仕組みにより、Bot開発者しか利用できないようになっています。ユーザーに使わせたくないコマンドは、このcogに入れます。

4.2.4 Setting

ユーザーからの各Guildの設定についてのコマンドを受け付け、その内容をDBに書き込む処理を行います。具体的には、読み上げ文字数上限や、名前を読み上げるかどうかなどを設定するためのコマンドが含まれます。Botの操作方法にかかわるprefixの設定などの一部の重要なコマンドは、discord.ext.commands.checkによるコマンドごとの権限設定により、Guildの管理者権限を持つユーザーしか使えないように設定しています。

また、Guildに関するDBのデータを削除するのもこのcogの役割です。Guildからキックされる等、Guildが切断されたというイベントを検知すると、そのGuildに関するデータ、たとえば登録された単語やprefixなどについての設定をすべて削除します。

4.2.5 UserVoice

ユーザーの声設定に関連するコマンドを含みます。update_voiceコマンドは、ユーザーの声設定を更新するコマンドです。また、声を設定されていないユーザーの声設定を生成するのもこのcogの役目です。声設定の生成は、ユーザーIDをseedとしてPython標準モジュール「random」を用いて行います。

4.2.6 Sub

これは他のcogとはすこし毛色の違うcogで、shovel_red、shovel_greenといったサブshovelのためのcogです。サブshovelは単独でも使えますが、メインshovelとは異なり、単語登録や声設定、Guild設定などのコマンドを備えていません。そこで、招待時や読み上げ開始時などのタイミングで、そのGuildにメインshovelがいるか確認します。いなかった場合、「このGuildにshovelを招待してくれたら、もっと便利に使えますよ」というメッセージ（図4.7）を送信します。

図4.7: メインshovelを招待するよう勧めるメッセージ

shovel_green BOT Today at 5:01 AM

Multi shovel

shovel(青)を招待すれば、**声の変更**や**単語登録**、**読み上げ詳細設定**など、多くの機能を利用できます。
もしこのサーバーにまだshovelがいないのであれば、こちらをクリックして、ぜひshovelを導入してください。

すでにshovelがいるサーバーにも、このメッセージが出ることがあります。

サーバへの招待方法、使い方は「Multi shovel」をクリック

4.3　shovelの機能を請け負うモジュール

4.3.1　モジュールとは

Pythonでは、クラスや関数の定義が記述されたファイルをモジュールと呼びます。ひとつのファイルにすべてを記述してしまうと、どこに何が書いてあるのかわからなくなるだけではなく、関係のない処理同士が思わぬ結びつきを持ってしまうおそれがあります。責務に応じて、適切な大きさのモジュールに分割しましょう。以下ではshovelのモジュールの責務や実装について述べます。

4.3.2　Connection

本モジュールの責務は、shovelを各Guildのボイスチャンネルへ接続させることで、Yomiage cogからの指示を受けて動きます。ボイスチャンネルへの接続には多様な要素があります。そのため、再現性の低い不具合が発生しやすく、多くの例外処理やユーザーへのメッセージが必要となります。そこで、「ボイスチャンネルへの接続」という単一の処理ではありますが、専用のモジュールにまとめました。ボイスチャンネルの接続に関わる要素には、以下のようなものがあります。

・読み上げを実施中か、そうでないか
・読み上げ中の場合、読み上げが行える状態か
・Guildの権限、Guildチャンネルの権限、Roleの権限、shovelの権限
・Guildのリージョン
・Discord APIの稼働状態

ここで取り上げただけでもわかるように、接続に関わる要因は多岐にわたるため、実際は起きてみないとわからない不具合も多々あります。その中でも極力ユーザーが快適に使えるよう、ひとつひとつの不具合に対応する「適切なエラーメッセージ」と「解決方法」を伝えるため、細かく場合分けを行っています。ユーザーに的確なエラーメッセージを伝えるのは、サポートにかかる時間を減らすことにもつながります。

4.3.3　OpenJTalk

本モジュールの責務は、Open JTalkによる音声合成を行うことです。Python標準モジュール「subprocess」を使い、OSコマンド「open_jtalk」を呼び出して音声合成を行います。音声合成に使う「open_jtalk」は、処理自体はすぐに終わるものとはいえ、メモリ消費は大きくなります。そのため、偶然にも数十の音声合成が同時に走ってしまうとメモリ不足に陥るおそれがあります。よって、同時に実行する「open_jtalk」の数を制限しています。

また、OSコマンドの呼び出しにはsubprocess.asyncio.create_subprocess_execメソッドを用いることで、OSコマンドインジェクション攻撃への対策をしています。

4.3.4 Speakable

本モジュールの責務は、ユーザーの投稿した文字列から、モジュールOpenJTalkに渡す文字列、つまり実際に読み上げるための文字列を作成することです。たとえば、「afk」を「りせき」と単語登録しているGuildで、「ちょっとafkします」という投稿があったとしましょう。このとき、「ちょっとafkします」という文字列をうけとり、「ちょっとりせきします」という文字列を返すのがこのモジュールの仕事です。

このモジュールの具体的処理は以下のようになっており、上から順番に実施します。処理が多岐にわたるので、複雑にならないよう、下記の箇条書きにあるような変換処理はひとつひとつ関数として独立しています。変換処理の例を、図4.8に示します。

・複数行のテキストを1行にする
・URLを「URL省略」という文字列に変換する
・Guildに登録されている単語について、「読み」に変換する
・Guild絵文字をIDに変換する
・通常の絵文字をキーワードに変換する
・記号を削除する
・英語をカタカナ読みに変換する
・「w」を「わら」に、「www...」を「わらわら」に変換する
・過剰な繰り返しを削除する
・規定文字数以上であれば、末尾を省略する

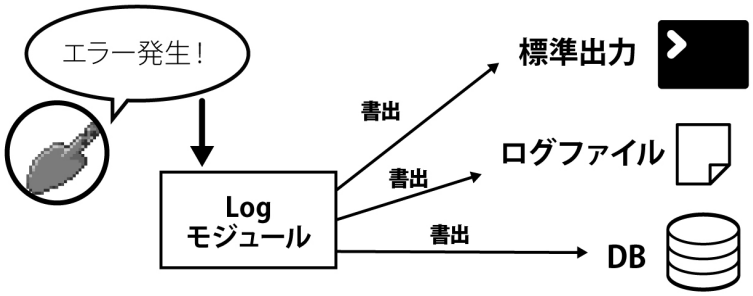
図4.8: 文字列変換の例

4.3.5 Log

本モジュールの責務は、3つの出力先（図4.9）にログを出力することです。ログの出力内容など、詳細は第6章「品質を上げるための設計のポイント」で紹介します。

1. 標準出力
2. ファイル
3. DB

図4.9: shovelの3つのログ出力先



標準出力、ファイル出力はPython標準ライブラリーのlogging.FileHandlerやlogging.StreamHandlerで実現可能です。DBへの出力を実現するために、独自クラスのDBLogHandlerがあります。DBLogHandlerは、logging.FileHandlerやlogging.StreamHandlerと同様にlogging.Handlerを基底クラスとするハンドラーです。このハンドラーはログを受け取ってDBへ書き出す処理を行いますが、ログを受け取るたびにひとつひとつ書き出すのは効率が悪いです。そこで、受け取ったログはハンドラー内部で保持し、一定時間ごとにまとめてDBへ書き出す方式をとっています。

shovelが使用するDBライブラリーであるpeewee3.8には非同期処理のインターフェイスがありません。DBへの書き出しには時間がかかる場合もあります。書き出しで処理が長時間止まってしまうと、Discordサーバーとの疎通確認やイベント処理などに影響するおそれがあります。なので、非同期処理としてDBへの書き出しを行うために、discord.Client.loopでBotのイベントループを入手し、loop.run_in_executor関数を通じてDB書き出し処理を呼び出しています。

4.3.6 Database

本モジュールの責務は、DBへの接続を行い、データの読み書きを行うことです。shovelはpeeweeeというORMを使用していますが、それをさらに包み、各モジュールに提供しているのがこのクラスです。機能を満たすだけであれば、各モジュールで直接peeweeeを操作してもまったく問題はありません。それどころか、わざわざモジュールDatabaseを通すことはコーディングの量も動作する際の処理の量も増え、煩雑になることを意味しま

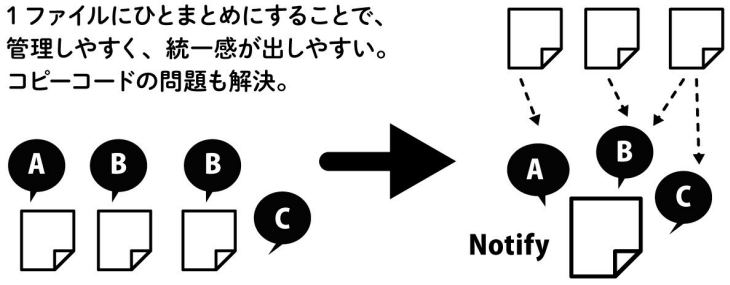
す。ではなぜこのようなモジュールを用意しているのでしょうか。ひとことで言えば、モジュール間の結合度を下げて保守性を向上するためです。詳しくは第8章「実装 - いざ、コーディング」の「保守性を高める実装の例」の節で詳しく説明します。

4.3.7 Notify

本モジュールの責務は、ユーザーに送信するすべてのメッセージの種別と内容（メッセージ文字列および出力する情報の種類）を保持することです。shovelは、ユーザーにエラーメッセージや処理完了メッセージなどさまざまなメッセージを送信します。shovelには多くのモジュールがありますが、それらのモジュールの中に直接メッセージ内容を書くことはしていません。メッセージ内容はモジュールNotifyが保持していますので、メッセージを出力したいモジュールはモジュールNotifyへメッセージ内容を取りにきて、それをそのまま送付します。

このような設計にしているのには理由があります。図4.10をごらんください。ユーザーに送信するメッセージの定義をひとつのモジュールにまとめると、すべてのメッセージを見通しよく管理することができ、フォーマットや文言を揃えやすくなります。また、異なるモジュールから同じような内容のメッセージを送る必要が生じたとき、コピーコード（同じコードが別の箇所に複数存在すること）の発生を抑制します。以上のような利点をふまえ、文言を包括的に管理する本モジュールを用意しています。

図4.10: メッセージを集約するモジュールがあるメリット



4.4　多くの外部パッケージがshovelを支えている

ここでは、shovelで使っている外部パッケージを紹介します。

4.4.1　外部パッケージを使う理由

本書において「外部パッケージ」とは、開発者が書くコードから利用するパッケージのうち、標準ライブラリーに含まれず、pipやダウンロード等で導入する必要があるものを指します。Pythonには便利で充実した標準ライブラリーが存在しますが、アプリケーションの仕様によってはそれだけでは不足していることもあります。そんなときは「車輪の再発明」に陥らないよう、外部パッケージの利用を検討しましょう。

4.4.2　外部パッケージ導入の注意点

外部パッケージは便利ですが、注意点もあります。パッケージにはライセンスが添えられていることが大半ですので、よく読んで理解したうえで使うようにしましょう。ライセンスがないパッケージは、開発者にライセンスを確認しましょう。ライセンスがないからといって、自由に使っていいわけではありません。また、使いたいパッケージのライセンスだけでなく、そのパッケージが依存するパッケージやアプリケーションのライセンスまで含めてしっかり確認する必要があります。

4.4.3　discord.py

PythonでDiscord Botを作る際にもっともよく使われているAPIラッパーです。特徴として、モダンなPythonインターフェイス、レートリミット制御、Discord API完全準拠、速度とメモリ効率の両立が挙げられます。

少ないコード記述量ですぐにBotを開発できるいっぽうで、非同期処理（async/await）やデコレーターを多用するので、プログラミング初心者が少々つっこんだことをすると詰まりやすいという面もあります。

discord.pyについては、第1章「DiscordとDiscord Bot」で詳しく説明していますので、そちらもごらんください。

4.4.4　peewee + PyMySQL

peeweeは、Python用ORMです。ORMは、DBのデータをオブジェクト指向プログラムで扱いやすくするための手法です。ORMには賛否両論ありますが、shovelでは書きやすさを優先し、採用しました。PythonのORMモジュールとしては高機能なSQLAlchemyも人気ですが、shovelにはシンプルなもので必要十分だったのでpeeweeを採用しています。

実際にshovelで使っているのはMariaDBですが、前述のとおりMariaDBはMySQLと互換性がありますので、PyMySQLを使います。こちらもMITライセンスでリリースされています。

4.4.5 Click

Clickは、Pythonアプリケーションにコマンドラインインターフェイスを組み込むためのパッケージです。通常コマンドライン引数を扱うためには複雑な記述が必要になりますが、Clickを使えば簡単に引数つきプログラムを作成できます。リスト4.1をごらんください。

リスト4.1: Click コードサンプル(click_sample.py)

```
= click_sample.py
import click


@click.command()
@click.option('--count', required=True, type=int)
@click.option('--name', required=True, type=str)
def say_hello(count, name):
    for i in range(count):
        print(f"こんにちは、{name}さん")


if __name__ == "__main__":
    say_hello()
```

このPythonプログラムは、リスト4.2のように使用できます。--helpオプションで表示される内容は自動的に生成されます。

リスト4.2: Click 動作サンプル

```
$ python3.7 click_sample.py --name cod --count 3<Enter>
こんにちは、codさん
こんにちは、codさん
こんにちは、codさん
$ python3.7 ./click_sample.py --help<Enter>
Usage: click_sample.py [OPTIONS]
Options:
  --count INTEGER  [required]
```

```
  --name TEXT      [required]
  --help           Show this message and exit.
```

shovelでは、リスト4.3のように、デバッグレベルの指定や、読み込む設定ファイルの指定に使用しています。

リスト4.3: Clickを利用したオプション指定付きでのshovelの起動

```
$ python3.7 launcher.py --level DEBUG --config_file config_red.py<Enter>
```

4.4.6 jishaku

discord.pyの開発・デバックを支援するライブラリーです。ある程度の規模のDiscord Botを運用される方ならぜひ導入するべきライブラリーです。詳細は、第10章「Discord Botのテスト自動化」にて紹介します。

4.4.7 alkana.py

アルファベットをカタカナ読みに変換するためのライブラリーです。たとえば、「Happy」を「ハッピー」に変換します。当ライブラリーは、筆者が開発し、PyPIに登録したものです。読み上げに際し英単語をカタカナ読みに変換したかったのですが、Pythonライブラリーでライセンスに不安のない方法が見当たらなかったため開発しました。

変換に用いているオリジナルデータは、PCの高度利用をバリアフリー化することを目的としたプロジェクト、BEPの辞書データであるbep-eng.dicです。

4.5　shovelが使う外部アプリケーション

4.5.1　外部アプリケーションを使う判断

アプリケーションを作る際には、とくに理由がなければ単一のプログラムで構成します。単一プログラム内ではデータのやり取りの記述がシンプルですし、メモリ上のやりとりで済むためスピードも出ます。もしここで、プログラムをメインとサブのふたつに分割した場合、データのやりとりにオーバーヘッドが発生し、プログラムが遅くなってしまいます。また、メインプログラムからサブプログラムを呼び出す構成にした場合、メインプログラムが終了した際にサブプログラムのプロセスを終了するなどの処理も必要になりますから、複雑になります。

ですが、それでもあえてプログラムを分けたほうが良いケースもあります。たとえば、CPU負荷の高い処理について、プログラムを分割して負荷の高い処理を他のプロセスに任せると、メインプロセスのCPU使用率を低く保ち、サービス本来の処理が滞る可能性を抑えられます。このとき、CPUがマルチコアであればパフォーマンス向上も期待できます。また、shovelにおけるOpen JTalkのように、使用したいアプリケーションが別プログラムとして提供され、別プロセスとして動作させることが要件として決まっている場合もあります。すべての機能を単一のプログラムで完結させる必要はありませんが、無理に分けることもありません。アプリケーションの要件に応じて、別のプログラムを呼び出すことを検討するとよいでしょう。

以下では、shovelが使う外部アプリケーションについて述べます。

4.5.2　Open JTalk

Open JTalkは、修正BSDライセンスでリリースされているオープンソースの日本語TTSアプリケーションです。コマンドラインから起動し、標準入力で音声合成対象の文字列を受け取ると、形態素解析ののち音声合成を行い、wavファイルを生成します（図4.11）。

図4.11: Open JTalkの動作

HTSファイルという音声データを与えることで声質を変更可能で、shovelではmeiという女性ボイス音源を利用しています。また、発話スピードや声のトーンも簡単に調整可能なので、shovelの特徴である多様な読み上げボイスの作成という点で大きな役割をはたしています。

4.5.3 ffmpeg

ffmpegは、LGPLまたはGPLでリリースされている動画や音声を記録・編集・再生できるオープンソースのアプリケーションで、豊富な機能が特徴です。

shovelでは、Open JTalkで合成した音声をボイスチャンネルへ流す前に必要に応じて編集するために使用しています。また、discord.pyもボイスチャット機能で音声再生やエンコードを行うためffmpegを使用しています。

第5章　大勢に使ってもらえるサービスを目指して

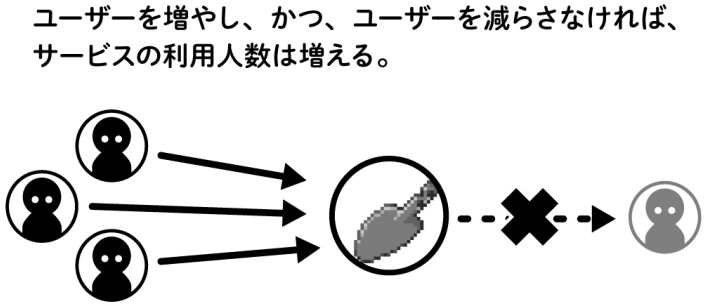
　この章では、個人開発というお金も時間も限られた条件下でサービスを作り、多くのユーザーに使ってもらえるようにするうえで、とくに重要な「品質」とそれを担保するための方法について述べます。また、shovelへの機能追加を例として、サービスの開発プロセスについて説明します。

5.1　たくさんの人にサービスを使ってもらうには

5.1.1　利用者数はどう決まる？

単純な計算ですが、増えたユーザー数から、減ったユーザー数を引くと、現在のユーザー数になります。ユーザー数を増やしたいのであれば、ユーザーを増やし、同時にユーザーを減らさないことにも気を配る必要があります。（図5.1）

図5.1: たくさんの人にサービスを使ってもらうには



5.1.2　ユーザーを増やす

サービスをたくさんのユーザーに使ってもらいたいのであれば、マーケティング分野で行われる基本的な分析からやってみるのがいいでしょう。たとえば3C分析をSWOT分析と組み合わせて、3つのC（ユーザー・競合・自サービス）についてSWOTを書き出してみるのもひとつの方法です。そこから、今後の方向性や戦略を考えてみましょう。

また、広報活動をするなかでの情報展開と周知を通じて、ユーザーの新規獲得も期待できます。すでにサービスを運営していて、ユーザーを増やしたいのであれば、サービス内容を随時改善し、その内容をユーザーに告知します。また、広報キャンペーンを打ってみるのもよいでしょう。

5.1.3　個人開発の経営判断は不合理でもOK！

通常の企業がサービス運営をするのであれば、自社の力が及ばない部分に原因があってビジネスとして成立しない場合は撤退の判断をする必要があります。ですが、すべてを自分ひとりで引き受ける個人開発では、不合理な状況があっても、責任の持てる範囲で自分のしたいことを最大限貫くことができます。これ

は個人開発のおもしろいところのひとつです。需要がなくても、競合がいても、あなたがそのサービスを作りたいと思ったなら作ってかまわないのです。

とはいえ、あまり人気がない分野において、すでに競合サービスが幅を利かせていたら、自分のサービスを使ってもらうのは難しいように思われます。確かに人が多く競合サービスもない分野のほうが、人気サービスになるのは簡単でしょう。しかし、諦めることはありません。人気がないジャンルでも、これから人気になるかもしれませんし、うまくいけば自分の手でそのジャンルの人気を出すこともできるかもしれません。それに、すでに広く使われている競合サービスがあったとしても、ユーザーがそれを使っている理由は「それしかないから」なのかもしれないのです。

5.1.4　ユーザーを減らさない

仮に100万人のユーザーがサービスに流入したとして、そのサービスがずっとダウンした状態だとしたらどうでしょうか。ユーザーはがっかりして離脱してしまい、そしておそらくはもう帰ってこないでしょう。これは極端な例ですが、どんなサービスでもユーザーの信頼を得られなければ、いずれ去られてしまいます。たくさんの人に使ってもらえるようになるためにはユーザーを増やすことも大切ですが、それと同じくらい、ユーザーを減らさないことも大切です。サービスがユーザーの満足を満たすということは、ユーザーを増やすうえでの必須条件です。

5.2　shovelの戦略 - ユーザーを逃がさない！

5.2.1　顧客満足度第一

前節と矛盾するようですが、筆者はshovelについてのマーケティング分析は行っていません。その代わりに、前節でいう「ユーザーを減らさない」という一点、つまり「サービスの品質」を重視して開発・運営を行っています。ユーザーのニーズをしっかり捉えた機能を備えたサービスを、高いレベルの品質で提供していれば、離脱は少なく、さらにユーザーも口コミで増えていくであろうという考えに基づいて、この戦略をとっています。

5.2.2　shovelの満足度を支える要素

shovelのユーザーは、ゲームやTRPG、作業通話や勉強会など、さまざまな用途で読み上げを必要としています。shovelはそれに対し、読み上げを聞いただけで誰の発言かわかるユーザー声設定、応答速度100ms未満での読み上げ、スリーナインクラス（99.98%）の稼働率、わかりやすいユーザーインターフェイスといったサービスを提供しています。

稼働率については、月あたりのダウン時間は5分以下を目標としています。その対策として、万が一読み上げが行えない状態になっていると、開発者である筆者にアラートがあがるようになっています。また、shovelの応答速度は常にサービス内部で計測しており、一定期間継続して閾値を超えつづけると、こちらもアラートがあがります。サーバーの死活監視についても同様です。

これらはすべて、システムの品質を高めようという意識によるものです。では、システムの品質とは一体何なのでしょうか。次節からはそれを見ていきます。

5.3　システムの品質について考える

5.3.1　FURPS+

システムの品質について深く理解するために、FURPS+というモデルを見てみましょう。FURPS+ は、ソフトウェアの品質を測定する指標のモデルであり、その名称は指標の頭文字に由来します。それぞれの指標は下記のとおりです。

・機能性（functionality）

システムが期待される機能を持っているかどうか。

・使いやすさ（usability）

操作方法やUIなどを含むシステムの挙動に一貫性があり、明快であるか。

・信頼性（reliability）

故障しづらいかどうか。故障した際、安全であるか。

・性能（performance）

要求への応答速度はどうか。

メモリ・CPU・ネットワーク・記憶容量等のリソース消費量はどうか。

・保守性（supportability）

機能の追加削除などの変更がしやすいかどうか。

・プロジェクト上の制約（plus constraints）

実装方法や、インターフェイスの制約はあるか。

5.3.2　とくに重視していること

筆者がサービスを開発・運営するうえでもっとも重視しているのは、「使いやすさ」「信頼性」「保守性」の3点です。まずは、この3つの指標をなんとなく頭にいれておいてください。

5.3.2.1　使いやすさ（Usability）

shovelは、ユーザーの期待したとおりに、または期待以上に動き、使いやすいサービスであること目指しています。これについては、第6章「品質を上げるための設計のポイント」で詳しくお話しします。

5.3.2.2　信頼性（Reliability）

ユーザーがshovelを使いたいと思ったとき、常にすぐに使えることを目指しています。このために、正常系だけでなく異常系も考え得る限りの洗い出しを行い、エラーを処理しています。異常系の洗い出しについては、第6章「品質を上げるための設計のポイント」で述べます。また、shovelに問題が発生したときや、サー

バーに故障があったときにも即座にサービスを復帰させるための仕組みを整えています。これについては、第14章「監視 - 24時間みまもり体制」をごらんください。その他にも、アップデート時にダウンタイムを最小化する工夫もしています。これについては、第11章「アップデートのための作業」で解説しています。

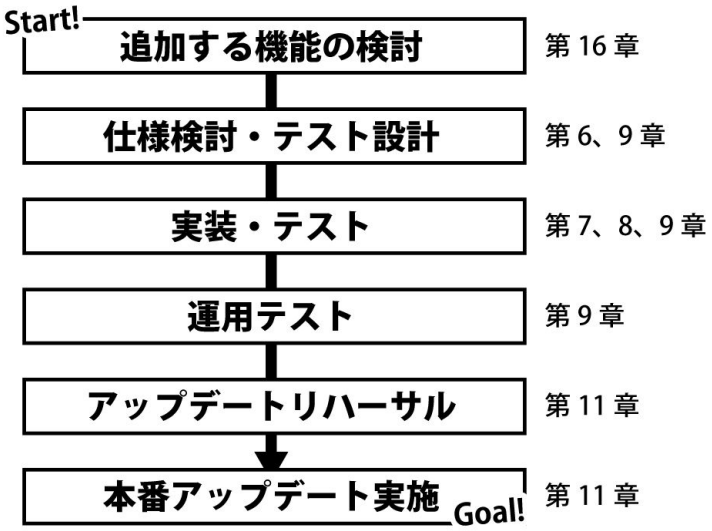
5.3.2.3 保守性（Supportability）

機能を追加削除する際に不便のないよう、普段からshovelのコードを整頓しておくことを心がけています。これについては、第8章「実装 - いざ、コーディング」で詳しく述べます。コードだけでなく、周辺要素（DBやWebサーバー等）についても同様です。

5.4　開発プロセスを俯瞰する

ここから先の第6章から第11章では、shovelに機能を追加する際、実際にどのような手順を踏んでいるかを具体的に紹介します。この章で述べた「品質」を担保するために各手順で行っているさまざまな工夫についてもお話しします。個別の手順の解説へと進む前に、まずプロセス全体を俯瞰してみましょう（図5.2）。

図5.2: 開発プロセス



5.4.1　追加する機能を決める

Botに追加する機能について検討し、決定します。このプロセスについては、第16章「運営 - ユーザーと接しよう」で詳しく触れます。

5.4.2　仕様検討・テスト設計

追加する機能が決まったら、機能の詳細な仕様を詰めていきます。仕様を検討しながら、同時にテスト設計も行います。この手順については、第6章「品質を上げるための設計のポイント」で詳しく紹介しています。ユーザーからの要望に由来する機能を実装する際には、この手順が非常に重要です。たとえば、「NGワードを登録したい」という要望があったとします。これは「ある単語を読み上げないようにしたい」であったり、

「NGワードを発言したユーザーに警告を出したい」であったり、さまざまな意味に解釈できます。あいまいな機能のイメージを、一意に解釈できる具体的かつ詳細な機能仕様に落とし込むのが本手順の目的です。

前述のとおり、この手順は、テスト設計と同時に行います。たとえば、「NGワードの登録は10個まで」という仕様が決まると、同時に「10個より多くのNGワードを登録できないこと」というテストを行う必要が出てきますので、仕様検討とテスト設計は表裏一体の関係といえます。これについては、第9章「テスト - コードの品質をまもる、最後の砦」でより詳しく触れます。

5.4.3 実装・テスト

決定した仕様に沿って実装します。動作確認は随時、開発用Botで行います。簡単な仕様であれば、一気にコーディングしてテストを行います。それに対し、複合的・大規模な機能追加の場合は、少しずつ開発・テスト・リファクタリングを繰り返しながら実装をすすめていきます。DBの構造変更が必要なケースでは、マイグレーション用のスクリプトの記述もここで行います。実装についての詳細は、第8章「実装 - いざ、コーディング」で紹介します。

目に見える不具合が取れ、設計したテストを実施し、致命的な不具合が発生していないことを確かめることができたら、運用テストに移行します。テストの実施については、第9章「テスト - コードの品質をまもる、最後の砦」で詳しく紹介します。

5.4.4 運用テスト

テスト用フォルダーのコードを最新版に同期し、テスト用Botをアップデートします。テスト用Botをしばらく（一晩程度のことが多いです）稼働させ、問題が発生しないことを確認します。ここではじめて自分以外のユーザーが操作するため、思いもよらない不具合やエラーが発見されることもあります。

5.4.5 アップデートリハーサル

DB構造など、重大な部分の変更を伴うアップデートである場合、アップデートの手順書作成を行い、リハーサルを実施します。アップデートのリハーサルと本番アップデートについての詳細は第11章「アップデートのための作業」で紹介します。

5.4.6 本番アップデート実施

本番環境のコードを最新版に同期します。リハーサルで行ったのと同じ手順でDB構造をアップデートし、本番環境Botをアップデートし、再起動します。この時点で不具合が発生することは稀ですが、念には念を入れ、ここでもテストを実施します。問題ないことがわかったら、ユーザーへのリリース通知を実施します。リリ

ース時、ユーザーに不便をかけないための工夫については第11章「アップデートのための作業」、リリース通知については第16章「運営 - ユーザーと接しよう」で詳細を紹介します。

第6章　品質を上げるための設計のポイント

この章では、サービスを作る上での設計のポイントについて述べます。まずは「使いやすさ」を向上させるための考え方について、具体例としてshovelの機能を例に挙げながら説明します。次に、「信頼性」を向上させるために有効な手段である「異常系」を見つける方法と、異常系への対応方法について述べます。最後に、サービスのログの重要性について、具体例を挙げながら説明します。

6.1　ユーザビリティを考える

6.1.1　ユーザビリティとは

ユーザビリティという言葉が指すものごとは大変多様なのですが、一言で表すならば「使いやすさ」のことだと筆者は考えています。この本では、「ユーザーがそのシステムを使いやすいと感じるかどうか」をユーザビリティと呼ぶことにします。

6.1.2　地味だけど重要なユーザビリティ

この問題の難しいところとして、ユーザビリティが優れていること自体を売りにしづらいということがあります。ユーザビリティが優れているシステムは、それをユーザーに感じさせません。ユーザーがしたいことを、したいようにできる。そんな透明さがユーザビリティ設計のめざすところです。ですので、ユーザビリティが優れているというだけでユーザーを増やせるわけではありません。

ですが、ユーザビリティが損なわれているシステムは、ユーザーにたえずストレスを与えます。どんなに優れた機能をそなえていても、ユーザビリティを疎かにしていては、その機能は結果的にユーザーを困惑させるだけの存在に成り果てます。たとえばshovelが人間のようになめらかな読み上げを行うBotであったとしても、漢字がまったく読めなかったり、読み上げるまでに5分のタイムラグがあったりするようでは、だれもshovelを使わないでしょう。

6.2　ユーザビリティの実例 - shovelの工夫

　ここでは、shovelのユーザビリティを高めるうえで工夫していることをいくつか実例として紹介します。参考にしてみてください。

6.2.1　ユーザーに送信するメッセージの工夫

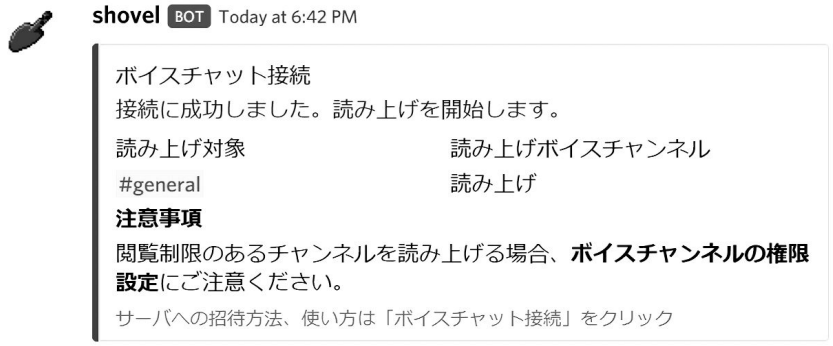
6.2.1.1　Botがユーザーに送信する文言はとても大切！

　サービスを開発する際、ユーザーに表示するメッセージの内容に気を配るべきなのは当然ですが、Botにおいてはことさら重要です。なぜなら、一般的なWebアプリケーションと異なり、Botは説明書や補足事項をユーザーに見てもらうことが難しいためです。外部に充実した説明書があってもユーザーがそれを知る手段はありません。「Botを導入したはいいけど、その後どうしたら使えるのかわからなかった」というのもよく聞く話です。また、わかりやすい文言は、サービス開発者がユーザーからの問い合わせに対応することによる時間的拘束を減らすということにも繋がります。

6.2.1.2　Embedを使ったわかりやすい通知

　shovelでは、ユーザーに通知するメッセージはすべて、Discord Embedというフォーマットで表示します。Discordを利用している方なら見覚えがあると思いますが、図6.1のようなものです。

図6.1: Discord Embedの例



　リンクなどを貼ったときに自動的にEmbed形式のものが表示されることはありますが、原則ユーザーが任意のEmbedを投稿することはできません。Botのメッセージとユーザーの投稿を明確に区別できるという点に着目し、shovelがユーザーに通知する内容はすべてEmbedにしました。

6.2.1.3　Embedには色を設定できる！

また、Embedを使うメリットはもうひとつあります。それは、Embedには色を設定できるという点です。図6.1の画像にあるように、メッセージを囲む枠の左に太い線があり、この色を自由に設定できます。shovelではこの色を用途に応じて図6.2のように使い分けています。

図6.2: Embed色の用途

名前	色	用途
SuccessNotify	blue	処理が正常に完了したとき
WarningNotify	orange	必要な条件が整っておらず、処理が実行できないとき
ErrorNotify	red	異常が発生し、処理が失敗したとき
NormalNotify	gray	処理の結果ではない、単なる情報の通知

このように使い分けるメリットは2点あります。1点目は、ユーザーが見たときにわかりやすい点です。一貫性をもって色を使うことで、はじめてshovelを使う人でも、メッセージの内容をよく読まなくても、「なにか失敗したな」「成功したな」と直感的にわかります。2点目は、開発者である筆者自身がメッセージの目的を考えながら記述できる点です。どの色を付けるべきかということを考えるのは、自分がいまユーザーに何を伝えたいかを具体的に検討することに等しく、曖昧で不明瞭になりがちなメッセージをわかりやすいものに改善できたのです。これは筆者にとっても思いもよらない効果でした。

6.2.1.4　Embedにはリンクを挿入できる！

他にも、図6.2のEmbedには最下部に小さな文字で一言書く「footer」という欄があります。そこに、「さらに細かい説明はリンク先にありますよ」と表示することで、操作方法がわからないユーザーを説明書に誘導できます。

6.2.2　色だけに頼らないUI（アイコン）

shovelには、メインのshovel「shovel」と、サブのshovel「shovel_red」「shovel_green」があります。これらはそれぞれ青、赤、緑のアイコンです。このうち、赤と緑には色覚バリアフリーであるよう「R」と「G」という文字を入れています。これによって、アイコンの色が判別しにくい方にも、一目でshovelを見分けられるようになっています。（図6.3）

図6.3: shovelRGB それぞれのアイコン

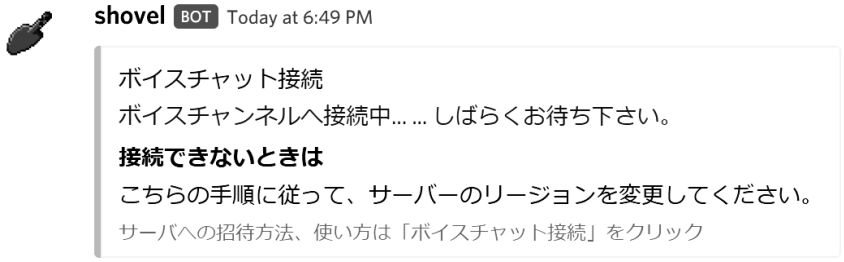
shovel（青）　shovel_red　shovel_green

6.2.3　時間がかかる処理をユーザーに通知

ボイスチャンネルへの接続には時間（短くて1秒未満、長くて数秒）がかかります。ユーザーがBotにコマンドを送ってからBotが反応するまでに時間がかかると、ユーザーは不安に思い、コマンドを連続で送信してしまうこともあります。これを防ぐため、接続を開始する前に接続中であることを示すメッセージ（図6.4）を送信します。このメッセージは、接続が完了、または接続に失敗すると削除されるので、ユーザーの邪魔になりません。

また、Discordのボイスチャット接続に関する不具合は、ボイスのリージョンを変更すると解決することもしばしばです。そこで、接続がうまくいかない際にはリージョンを変更することを薦めるメッセージも一緒に出しています。

図6.4: 接続中であることを示すメッセージ



6.2.4　投稿から読み上げまで0.1秒！

shovelがオープンベータ化して2週間ほどのことです。夜間のピークタイムに読み上げ対象Guildが数百に達すると、読み上げるまでに1〜5秒程度の遅延が発生するようになりました。これは、ゲーム中のコミュニケーションにはとても実用に耐えないと感じる数値です。ゲーマーのユーザーが多いshovelにとってこの遅延は大きすぎる、早急に対応する必要がある、と筆者は判断しました。

さっそく動作詳細ログを調査したところ、読み上げ音声を合成する処理がボトルネックとなっていることがわかりました。その処理はOpen JTalkプロセスを起動して実施されるのですが、第4章「shovelのソフトウェア構成」で述べたとおり、Open JTalkプロセスの同時実行数には上限を設けてあります。この遅延が発生した当時、同時実行を1プロセスまでに限定しており、そのために処理が追いついていなかったのです。

本処理は数理工学の初歩の待ち行列理論モデルで近似できます。混雑している時に窓口を増やすと劇的な待ち時間削減効果を見込めることがわかっています。必要なリソースを計算したうえで、3プロセスまで同時実行可能に変更しました。結果として、遅延は0.1秒以下になりました。

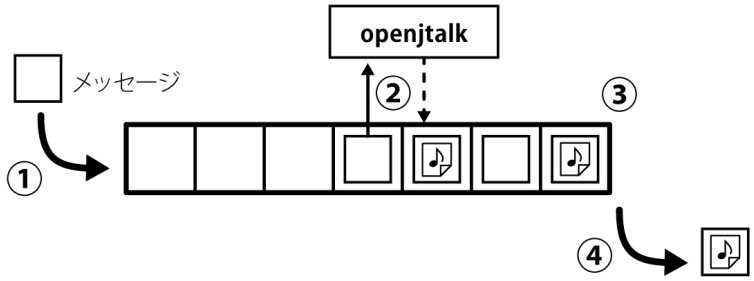
この遅延時間はユーザビリティに直結するものであり、リソース拡張の判断にも関わってくるものです。現在も常に監視し、1秒を超え続けるとアラートが上がるように設定しています。

6.2.5　読み上げ順序の保証

shovelの読み上げ機能で特徴的なのが、「必ずメッセージをshovelが受信した順に読み上げを行う」という点です。読み上げは、まず音声合成を行い、その結果を再生する、というステップを踏んで行われます。音声合成は文章が長いほど時間がかかります。単に「音声合成が終わり次第どんどん再生する」方法をとってしまうと、とても長い文章のあとに、とても短い文章が来た場合、あとに来た文章が先に読み上げられてしまいます。長文のメッセージのあとに、「うん」というメッセージが投稿された場合に、「うん」が先に読まれてしまっては、画面を見ていない人は混乱します。読み上げる順番が正確であることは、読み上げBotとしては必須の機能といえるでしょう。

ここで、メッセージを受信した順番の通りに読み上げを行うための仕組みを紹介します。まずGuildごとに配列を用意します。次に、音声合成を行う前に、その配列にメッセージIDを入れます。最後に、音声合成が終わったら、配列の先頭が自分のIDと一致するまでsleepを繰り返します。これにより、メッセージの長短にかかわらず、順に音声を再生することが可能になります（図6.5）。

図6.5: 読み上げ順序が保証されるしくみ



1. キューに並ぶ
2. 音声合成を行う
3. キューの先頭に到達し、かつ音声合成が完了していたら音声再生を開始する
4. 音声再生が終わったらキューから抜ける

6.2.6　音声合成用文字列のこだわり

投稿から読み上げ文字列への変換は、ユーザーからの要望も多様な部分であり、Botの使い心地にも直結します。工夫している点について、いくつか紹介します。

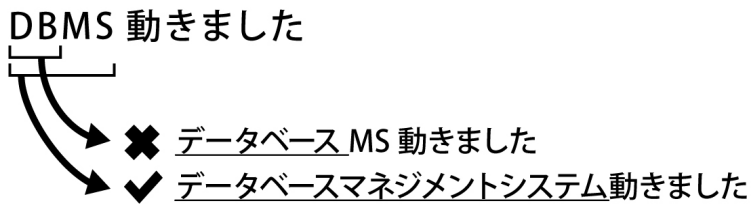
6.2.6.1　ユーザー登録単語が最優先

読み上げ文字列の作成は、大きく分けて2種類の変換によって行われます。ひとつはshovelがあらかじめ備えているデフォルトの変換、もうひとつはユーザーがGuildごとに登録した単語による変換です。変換は、後者を優先して実施し、デフォルトの変換処理がユーザー登録単語の邪魔をしないようにします。

6.2.6.2　より長い単語からマッチ

単語は、より長いものからマッチさせます（図6.6）。たとえば、「DB」が「データベース」、「DBMS」が「データベースマネジメントシステム」と登録されているGuildがあるとします。ここで「DBMS動きました」というテキストメッセージの読み上げが「データベースMS動きました」とならないよう、「DB」よりも先に「DBMS」を先にマッチさせます。

図6.6: より長い単語からマッチする

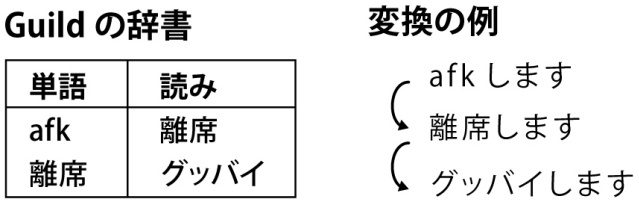


6.2.6.3　再帰的な単語マッチ

単語の「読み」にさらに別の単語が含まれているときも、なるべく再帰的に変換します（図6.7）。

たとえば、「afk」が「離席」、「離席」が「グッバイ」と登録されている場合、「afkします」という投稿が「グッバイします」と読まれるよう、再帰的に変換を行います。ただし、無限ループに陥らないよう、回数には上限を設けています。

図6.7: 再帰的な単語マッチ



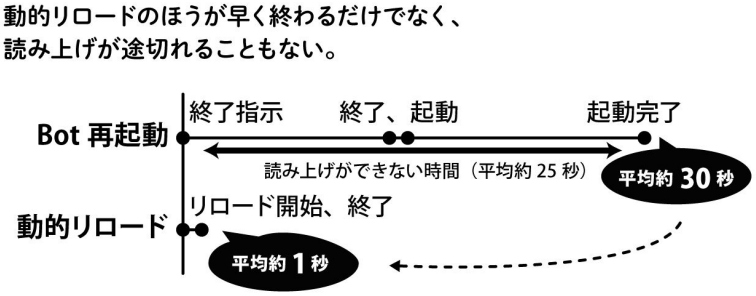
6.2.7　ユーザーに負担の小さい再起動

shovelは、アプリケーションを再起動しても、ボイスチャンネルへの接続状態を損ないません。もちろんアプリケーションを終了しているあいだの読み上げを行うことはできませんが、起動時に前回の接続状態を復元するようなつくりになっています。開発上・運営上の都合によりBotを再起動せざるを得なくなった際にも、十数秒のあいだ読み上げが途切れる以外の影響をユーザーに与えることがありません。この仕組みについては、第4章「shovelのソフトウェア構成」で解説しています。

6.2.8　ユーザーの使用を中断しない更新

再起動時にユーザーの接続状態を復帰する機能があるとはいえ、一時的にでも読み上げが途切れるのは望ましくありません。少しでもBotの再起動回数を減らすため、可能な限りモジュールのリロードによるアップデートを行うようにしています（図6.8）。モジュールのリロードを実装する方法については、第8章「実装 - いざ、コーディング」をごらんください。

図6.8: Bot再起動と動的リロードの比較



6.2.9　短いダウンタイム

ダウンタイムはそのまま、サービスをあてにしてくれているユーザーに不便をかける時間です。ダウンを伴う作業は、ダウンタイムができるだけ短くなるように手順を工夫しましょう。

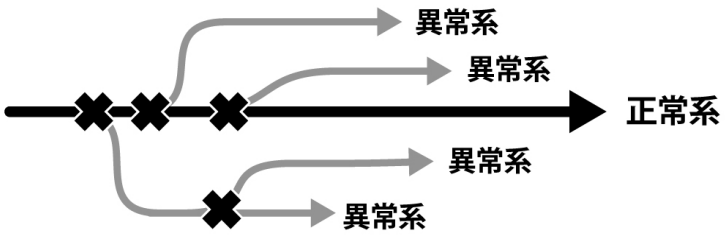
少し興味深い計算をしてみます。shovelのダウンタイムは、月間平均5分ほどであり、稼働率が99.98%であることはすでに述べました。これはアップデートに伴う再起動や、サーバー移行作業の所要時間などをすべて含む数字です。いっぽう、shovelのユーザ数は50万人。ここで、ダウンタイムに全員がアクティブだったと仮定して、ダウンタイムとユーザ数を掛け合わせてみます。すると、のべダウンタイムはなんと40,000時間、1,736日にものぼるのです。0.02%のダウン時間を多いと考えるか、少ないと考えるかは人それぞれですが、筆者自身はもっと減らしていかなければならないと感じています。ダウンタイムを短くするための工夫については、第11章「アップデートのための作業」で紹介しています。

6.3　異常系に漏れなく対応しよう

6.3.1　正常系と異常系

プログラミングにはエラーがつきものです。ここでのエラーとは、「プログラムが指示を処理している間に起きる障害」という意味です。エラーが発生した場合の動作を、「異常系」の動作と呼ぶことにします。対義語は「正常系」の動作、これは最初から最後まで問題が起きることなく処理完了するパターンの動作です。（図6.9）

図6.9: 正常系と異常系



バグの少ないプログラムとは、システムの状態や入力について多くのパターンを想定して、異常系の各ケースを洗い出し、それらに適切に対応できているプログラムといいかえることができます。shovelを開発する際は、信頼性とユーザビリティを高めるため、異常系を極力見逃さないことに心血を注いでいます。

では、どうすれば異常系を漏れなく見つけることができるのでしょうか。これについて考えていきましょう。

6.3.2　異常系の洗い出しテクニック

ここでは、異常系を漏れなく発見するための手順についてお話しします。

6.3.2.1　題材：読み上げ上限文字数の設定機能

異常系を洗い出す手順を説明する題材として、shovelの「読み上げ上限文字数の設定」(図6.10)について考えてみましょう。これは、shovelが投稿を読み上げるときの文字数の上限を!sh read_limitコマンドで設定するという機能です。読み上げ文字数上限を100に設定すると、100文字目以降はすべて「以下略」に置き換えられ、読まれなくなります。

図6.10: 読み上げ上限設定処理の例

6.3.2.2　登場する要素を書き出す

目的に関係する条件と、ユーザーが行う操作に、どのようなものがあるかをすべて書き出します。読み上げ文字数の設定に関連する要素を洗い出しましょう。項目の洗い出しは、まずは正常系の処理の流れを追うことで行います。お話を追いながら登場人物を羅列するイメージで、出てくる要素をすべて書き出します。要素を書き出すと、以下のようになります。

1. ユーザーの権限
2. ユーザーの入力＝読み上げ上限文字数
3. DB＝読み上げ上限文字数の保存先
4. 結果通知先チャンネル

6.3.2.3　各要素についてパターンを検討する

次に、この各要素について、起きうる事態、取り得る値のパターンを書き出します。ひとつひとつの要素について順番に検討しましょう。書き出した結果は以下のようになります。

ユーザーの権限

1. ユーザーにコマンドを操作する権限がある
2. ユーザーにコマンドを操作する権限がない

ユーザーの入力

1. 正常な値
2. 数値以外の値
3. 下限以下の値
4. 上限以上の値

DB

1. 書き込みに成功する
2. 書き込みに失敗する
3. 接続不良が起きている

結果通知先チャンネル

1. 通知に成功する
2. 通知に失敗する

それぞれの要素の1番目のパターンは、「正常系」の動作のパターンです。

6.3.2.4　要素同士の掛け合わせが必要か検討する

要素同士に関連のあるもの同士のパターンを掛け合わせることで、すべての異常系動作を洗い出すことができます。関連のない、または意味のない要素同士について掛け合わせることはありません。今回であれば、「ユーザーにコマンドを操作する権限がない」場合は、それ以降のコマンドの内容を実行することはないので、「ユーザーの権限」のパターンを「読み上げ上限文字数」や「DB」のパターンと掛け合わせる必要はないでしょう。しかし、「ユーザーにコマンドを操作する権限がない」場合、その旨を「結果通知先チャンネル」に通知する必要があります。つまり、「ユーザーの権限」のパターンと「結果通知先チャンネル」のパターンに関しては、掛け合わせが必要になります。図6.11のように表を描いて、このような掛け合わせの必要性を整理します。

図6.11: 各要素の掛け合わせが必要かの検討

	ユーザーの権限	ユーザーの入力	DB	通知先チャンネル
ユーザーの権限	-	-	-	-
ユーザーの入力	×	-	-	-
DB	×	×	-	-
通知先チャンネル	○	○	○	-

6.3.2.5　表を使ってすべての異常系を書き出す

こうして、それぞれの要素同士の関連の有無が検討できました。単純な機能ですが掛け合わせると膨大な数になり、すべて書き出すのはとても大変な作業だという予測がつきます。そこで、このようなパターンをすべて網羅するためには、表を使ったパターン洗い出し（図6.12）が便利です。

図6.12: 読み上げ文字数の設定 正常系・異常系全パターン

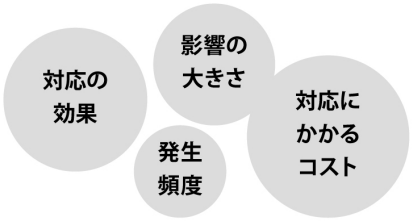
要素	値	1	2	3	4	5	6	7	8	9	10	11	12	13	14
ユーザーの権限	権限あり	○	○	○	○	○	○	○	○	○	○	○	○		
	権限なし													○	○
ユーザーの入力	正常な値	○	○							○	○	○	○	-	-
	数値以外			○	○									-	-
	下限以下					○	○							-	-
	上限以上							○	○					-	-
DB	書き込みに成功	○	○	-	-	-	-	-	-					-	-
	書き込みに失敗			-	-	-	-	-	-	○	○			-	-
	DB 接続不良			-	-	-	-	-	-			○	○	-	-
通知先チャンネル	通知成功	○		○		○		○		○		○		○	
	通知失敗		○		○		○		○		○		○		○

この表には合わせて14のパターンがあり、1が正常系、2〜14が異常系となっています。また、1〜14と書いてある列のひとつひとつがパターンをあらわしています。たとえばパターン1は、「ユーザーにコマンドを操作する権限がある」「正常な値」「書き込みに成功する」「通知に成功する」というパターンです。このように、表を使って組み合わせを確認することで、パターンが網羅できているかの見通しがよくなり、把握もしやすくなります。

6.3.3　異常系への対応を考える

異常系が洗い出せたら、これらひとつひとつについてどのように対応するかを検討します。対応といっても、「異常が発生した」というログを出すだけでもかまいませんし、場合によっては「なにもしない」という対応になることもあります。異常系への対応を考える際には、各異常系についてのリスク評価を実施し、対応を決定します。つまり、その異常系の発生頻度、異常系による影響の大きさ、対応にかかるコスト、対応したことで軽減できる影響、それらを見比べて、どのような対応をするか決定しましょう（図6.13）。

図6.13: 異常系への対応を検討する



たとえば「DBへの書き込みに失敗する」という異常系は、発生頻度こそ低いものの、ユーザーの要求した指示が果たせないため、影響は大きくなります（図6.14）。よって、「異常でデータが登録できなかったの

で、リトライしてみるようユーザーに通知する」という対応をすることに決めるのです。

図6.14: DBへの書き込み失敗時の対応を検討する

6.4　漏れた異常系を後からみつける

前節の手順で、「読み上げ上限文字数の設定」のような単純な機能ですら、異常系すべてを洗い出してみると膨大な数になることがわかりました。異常系を完璧に洗い出し、その対応も決め、そのとおりに実装したとつもりでいても、まだ検討できていない異常系は隠れているかもしれません。そのような異常系を見つけるための方法を紹介します。

6.4.1　漏れた異常系を見つける方法1 - 実際に使ってみる

Botを動作させ、自分の手でいろいろな操作をひととおり実行します。前述の方法であぶりだしたパターンをすべて試し、想定どおりの動きになっていることを確かめてみるほか、気の向くままに思いつきで操作してみます。手を動かしていると、他にも検討すべき条件や操作があることに気づくかもしれません。そのときは条件や操作の項目を追加し、あたらしいパターンについて検討しましょう。これは一般に「モンキーテスト」と呼ばれている手法です。サルが操作するかのごとく、適当に操作して不具合や見逃しをあぶりだすのです。

6.4.2　漏れた異常系を見つける方法2 - 大勢にテストしてもらう

自分ひとりでは限りがありますので、実際に使ってテストしてもらうのはもっとも有効な異常系のあぶり出し方法です。ユーザーが多ければ環境もその数だけあり、「そうきたか！」というような不具合をたくさん見つけることになるかもしれません。

とはいえ、テストしてくれるユーザーは、あくまでサービスに価値があるから使ってくれているというだけです。品質に問題があるBotは、Guildからキックされて終わりです。それだけで済むのであればまだよいのですが、Botの不具合でGuildに迷惑をかけることがあっては一大事です。オープンなテストを実施する際は、まず自分の中で完璧といえる状態にしてから提供するようにしましょう。

6.4.2.1　人海戦術への過信は禁物

shovelで実際にあった事例を紹介します。あるとき、「DMによるコマンドに反応しない」という不具合が発生しました。この不具合は機能追加によるアップデートのなかで埋め込まれ、アプリケーションログを確認したことで気付くまでの数日のあいだに数十件発生していました。この間、ユーザーからの報告はありませんでした。

不具合が発生したり、サービスがうまく動かなかったからといってユーザーが必ず報告してくれるわけではありません。大抵の場合は不便だなと思っておしまいです。ですので、不具合が発生したことをアプリケーション

ログなどから発見できる仕組みを作っておくことが重要です。また、不便をかけたにもかかわらず自分の時間を使って報告してくれるユーザーはたいへん貴重です。丁寧にお礼を伝えましょう。

6.5　ログはBotを良くするヒントの宝庫！

6.5.1　ログの意義

なぜログが必要なのでしょうか。まず、Botに不具合や異常が発生したときに何が起きたのかを調査するために役立ちます。どのような利用状況で、どのような操作をし、どのような結果になったのかということが残っていれば、不具合の原因を推測しやすくなります。次に、想定外の不具合が発生した際、ログによって気づくこともあります。これらはトラブルを未然に防ぐことや、万が一ダウンしても、そのダウンタイムを短くすることに寄与します。また、ログを確認することで、使われている機能・そうでない機能を知ることもできます。ユーザーの操作に対して異常が多く出ているコマンドは、使い方がわかりにくいコマンドであるとわかります。そのようなログを見つけ出して機能を改善することで、ユーザビリティを高め、問い合わせを減らすことができます。ログの活用については、第16章「運営 - ユーザーと接しよう」で詳しく紹介します。

このように、ログはBotを改善するための情報の宝庫です。もちろん、情報の宝庫にするためには適切なログを過不足なく出すことが必須です。では、どうすればそのようなログを出力できるのでしょうか。ここからは、それについて見ていきましょう。shovelでは、大きく分けて2種類のログを出力しています。

6.5.2　アプリケーションログ

アプリケーションログは、機能が動作したことや動作の結果を、ログレベルとともに自由な文字列で残すログです。例をリスト6.1に示します。アプリケーションログは、いわばBotの動作日誌のようなものです。

リスト6.1: ログ例

```
08/18 02:16:13 [INFO    ] MainThread cogs.yomiage   ( 450) : shovelがボイスチャンネルから退出
08/18 02:16:13 [INFO    ] MainThread speechconf     (  80) : キャッシュから各種設定を削除
08/18 02:16:13 [INFO    ] MainThread speechconf     (  82) : 削除完了
```

6.5.3　動作詳細ログ

アプリケーションログが「ログレベル」と「ログ文字列」という形式であるのに対し、動作詳細ログは内容に準拠した本システム独自のフォーマットで出力します。アプリケーションログはすべてDBの「ApplicationLog」というテーブルに格納します。それに対し動作詳細ログは、読み上げログは「YomiageLog」、ユーザー通知ログは「NotifyLog」といったように、それぞれ独自のテーブルに格納します。

6.5.3.1　動作詳細ログの保存期間

動作詳細ログはその性質上、Botを稼働させた時間とユーザー数の両方に比例して容量が大きくなっていきます。そのため、著しく容量が大きくなるような詳細なログは、期間を区切って削除または移動（アーカイブ）するようにするとよいでしょう。

たとえばshovelでは、読み上げ詳細ログは毎日集計して日次読み上げログとして別のテーブルに記録し、読み上げ詳細ログ自体は2週間以上前のものは削除するようにしています。

6.6　アプリケーションログはBotの活動日記

6.6.1　どんなときに出す？

アプリケーションログは、アプリケーションの流れを追うためのものです。ですから極端な話、アプリケーションのすべての経路でアプリケーションログを出してもよいのです。しかし、それだとあまりにも量が多くなり、意味のあるログが埋もれてしまいます。そこで、基本的には下記のようなときにアプリケーションログを出すとよいでしょう。

6.6.1.1　異常が発生したとき

実際にトラブルが発生して、問題判別のためにログを調べることを考えてみてください。あなたはそのトラブルの引き金になる異常をログから探そうとするはずです。もし想定内の異常だからといってログを残していなければ、何の記録も残っておらず、調査の手がかりが失われてしまいます。想定内であっても、異常のログを残しておくことはとても重要です。そうすれば、ユーザーからの問い合わせに対して、その動作は想定されたものである旨と対応方法について回答することができるでしょう。

では、想定外の異常についてはアプリケーションログを残さなくてもよいでしょうか？　もちろん、残せるのであれば残したいところです。しかし、想定外の異常はその性質上、意図的に残すことは不可能です。ですが、想定外であっても例外などを適切に捕捉し、少しでも取りこぼしがないように扱いたいものです。

6.6.1.2　低頻度の定期処理を行ったとき

開発者専用コマンドが実行されたときの処理や、月替り時の処理、1日1回の処理など、低頻度の処理については、開始と終了をログに出力しておいてもよいでしょう。「低頻度」という表現は曖昧ですが、重要なログを押し流すほどの物量であるかどうかを基準に検討すると良いでしょう。とくに、ユーザーと開発者、どちらにも処理の様子が見えないバッチ処理は、たとえばパフォーマンス低下が発生していても気づきにくいものです。処理の様子をログに残しておき、ときどきチェックしましょう。

6.6.2　アプリケーションログレベル

shovelはPython標準モジュール「logging」に倣い、DEBUG、INFO、WARNING、ERRORという4段階のログレベルを採用しています。Botを稼働させる際は、通常INFO以上のアプリケーションログを出力しています。DEBUGを出力すると、物量が多すぎるからです。

ログには、それぞれ適切なアプリケーションログレベルを割り当てましょう。こうしておくことで、時間がないときにも必要なアプリケーションログだけを閲覧できるようになるはずです。shovelでは、図6.15のようにアプリケーションログレベルを使い分けています。

図6.15: アプリケーションログレベルの用途

Level	用途
DEBUG	普段見なくてもいいような詳細な情報、INFO 相当だが数が多いもの
INFO	定常処理等の Bot の通常動作
WARNING	処理を継続できる想定内の異常の発生
ERROR	処理を継続できない致命的な異常の発生

6.6.3　出してはいけないアプリケーションログ

アプリケーションログはサーバー上に置き、Bot開発者しか見られないようになっています。しかし、何かの原因で流出するなどの事故の際にも致命的な事態を招かないよう、アクセストークンやBotの情報をアプリケーションログに出力しないことを徹底してください。

また、ユーザーの個人情報や投稿したメッセージの内容など「Botは扱う必要があるが、たとえBot開発者であっても閲覧不可能であるべき情報」もあります。このような情報についても、うっかりアプリケーションログに出すことがないよう、細心の注意を払いましょう。見てないから大丈夫、では通用しません。「起こりうることは起きる」と考え、どうやっても見られないようにしておく必要があります。

6.6.4　アプリケーションログの出力先

shovelでは、「標準出力」「ファイル出力」「DBへの出力」という3つの出力先へとアプリケーションログを出力しています。標準出力は、リアルタイムにアプリケーションログを閲覧するときに使います。本格的に監視を行うというよりは、正しく動いているかどうかを眺めるイメージです。ファイルは、Botの全体的な稼働状況をあとから時系列に沿って確認したいときに使います。

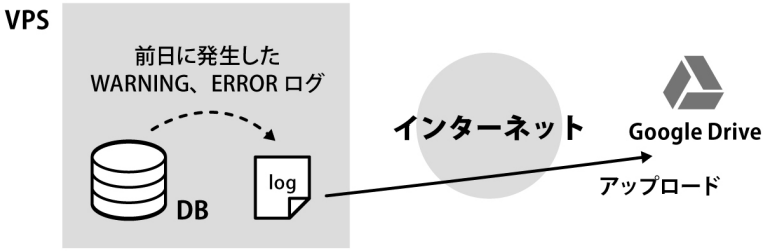
アプリケーションログを活用するためにもっとも重要なのが、DBへの出力です。DBに格納することで、統計的な観点からアプリケーションログを活用できます。また、Webインターフェイス等、外部へ開示する際にも手軽に扱えます。どのように活用するかについては次項で見ていきます。

6.6.5　アプリケーションログの活用方法

せっかく出力して蓄積しているログなので、有効に活用したいものです。shovelにおけるログの活用例をご紹介しましょう。

まず、ダイジェストの作成です。毎朝、前日のDBの内容のうち、WARNING、ERRORレベルのログを抜き出し、csv形式にしてGoogle Driveにアップロードしています（図6.16）。これはcronでシェルスクリプトを動かすことで実装しています。こうしておけば、開発者がDBにアクセスしなくても、重要なBotのアプリケーションログを簡単に確認できます。出先でSSHログインできないが、早急にログを確認したいときなどに活用できます。

図6.16: ログのダイジェスト作成



つぎに、Munin Pluginによるアプリケーションログ出力状況のグラフ化です。WARNING、ERRORレベルのアプリケーションログが何件発生しているかという数値をグラフに出力することで、著しく異常件数が伸びているときなど、Botになにか問題が発生した場合にひと目でわかります。また、Muninによるデータ蓄積により、継続的な発生件数の推移も確認できるため、Botの品質を監視できます。

第7章　開発環境 - 開発効率と品質をあげる礎

この章では、Botを開発するために必要な環境作りについてお話しします。ここでの「開発環境」とは、開発サイクルを円滑にまわすことに関わるすべてのことを指します。具体的には、以下についてお話しします。

・コードを書くエディター
・書いたコードを実行するための実行環境
・開発・テスト・本番の環境切り替え
・開発・テスト・本番環境でのコードの同期法

7.1　コードを書き、動かす環境を整えよう

7.1.1　コーディングと実行は開発の基本単位

コードを書いて実行する、というのはもっとも小さく基本的な開発のサイクルです。このサイクルを快適にまわせる環境を整えましょう。以下の3点さえ満たしていれば、どのような環境でもかまいません。

1. コードがかきやすく、誤りに気づきやすい
2. すぐ実行できる
3. 作業ミスが起きにくい

たとえば、コードを書いたあと動作確認をするために、毎回手作業でコードをサーバーにアップロードすることが必要なようでは問題です（2と3に違反）。コーディング中の誤字（typo）に実行するまで気付かないような環境も望ましくありません（1に違反）。この条件を満たすのであれば、IDEでも、テキストエディターでも、好みのものを使ってかまいません。

7.1.2　shovelの場合 - 開発環境は雲の上

shovelを開発する際は、運営用のサーバーにSSHログインし、テキストエディター「vim」でコーディング後、コマンドラインからPythonを実行して動作確認しています。この環境のメリットはふたつあります。1点目は本番とほぼ同じ環境での動作確認がすぐに行える点です。Gitなどのツールを使えばコードの同期はすぐ行えるとはいえ、コード変更とデバッグをくりかえし行う際には、手順はひとつでも減らしたいものです。2点目は、外出時などにBotにトラブルが発生したとき、SSHログインができるPCさえ用意すれば、すべてがいつもどおりの環境を使えるという点です。コードの修正が必要となったとき、エディターがいつもと違うとさらに思わぬミスを埋め込んだりする可能性がありますが、shovelはサーバー上のvimで開発しているため、そのようなリスクを減らせます。

デメリットとしては、開発環境を動かすことにより、肝心のshovelが使えるスペックを圧迫する可能性があるという点です。今の所、sshdとvimの使用メモリをあわせても、Bot自体の使うメモリと比較して著しく小さいので、この点についてはほぼ無視できるといって良さそうです。

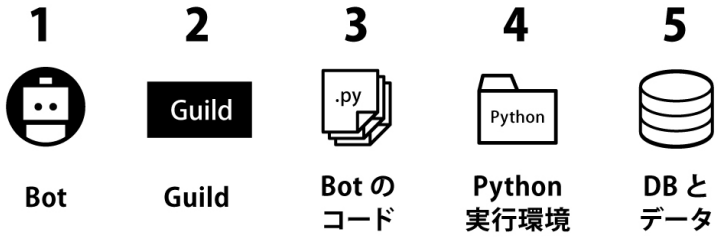
7.2　環境を3系統用意しよう

7.2.1　環境の5要素

Botを動かすのに必要な要素のセットを「環境」と呼びます。ここでは、環境についてわかりやすく説明するために、shovelを例として説明します。shovelをあなたのBotやシステムに置き換えながらお読みください。

shovelがサービスとして成立するために必要な要素は図7.1の5つです。

図7.1: shovelに必要な5要素



・サービスのインターフェイスとなるBot

・そのBotを自由に使えるGuild

・shovelのコード

・Python実行環境（インタプリタ本体と外部パッケージ一式）

・DBとその中身のデータ

このほか、ネットワークやOS含むサーバーが必要ですが、shovelではすべての環境を同じサーバーで構築しているため、一覧からは省いています。

7.2.2　3つの環境

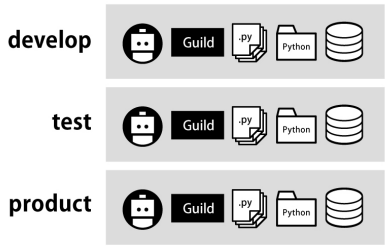
shovelは、前述の5要素がすべて揃った「開発（develop）環境」で開発しています。規模は小さいながら、サービスとして成立するための環境を作り、その中で開発しているわけです。ユーザーに公開するための「本番（product）環境」も同じ5つの要素が揃っています。ただし本番環境のGuildは、専用に用意したものではなく、利用者であるユーザーの管理するそれぞれのGuildです。

開発、本番のほかにもうひとつ、「テスト（test）環境」を準備しましょう。開発環境には最低限のデータしかなく、十分なテストが実行できません。かといって、すでにユーザーが利用しているBotを開発者の都合で

たびたび止めるわけにもいきません。shovelでは開発系で実施可能な範囲のテストをした後、もう少し本番に近い環境で、統合テストや少人数の運用テストを実施して本番運用に備えます。そのためのテスト環境を作り、そこでテストができるように準備しています。

　shovelでは、と言いましたが、どんなシステムでも常設の環境として最低でも「開発（develop）」「テスト（test）」「本番（product）」の3系統を稼働させるのがよいでしょう（図7.2）。

図7.2: 3つの環境



　この他に、たとえば手順の複雑なアップデート作業のためのリハーサル環境を作り、リハーサルだけ実施して廃棄することもあります。必要に応じて他の環境を作ることも選択肢に入れましょう。

7.3　環境をつくる手順

7.3.1　テスト環境を作ってみよう

各環境はデータが混じったり、実行するコードが混じったりしないよう、しっかり分離されている必要があります。ここでは、各環境の分離を確保しながら構築する方法について、テスト環境を作る場合を題材として記します。テスト環境を題材にしてはいますが、開発環境を作る際にも同じ手順を踏みます。

繰り返しになりますが、ひとつの環境を構築するために必要な要素は次の5つです。

・Bot
・Guild
・コード
・Pythonの環境
・DB

順に見ていきましょう。

7.3.2　テスト用Botの作成

すでに述べたように、shovelには「開発」「全体公開前の運用テスト」「本番」のそれぞれの段階に対応する環境が用意されており、それぞれの環境に対応するBotたちが存在します。図7.3をごらんください。

図7.3: 各フェーズに対応するBot一覧

Bot 名	用途	導入 Guild
shovel_develop	開発	開発用 Guild
shovel_test	全体公開前の運用試験	身の回りの少数の Guild
shovel	実運用	不特定多数の Guild

これらのBotは完全に独立したBotとして動作します。Botアカウントが違うのはもちろん、DBも独立しています。

7.3.3　テスト用Guildの作成

各Botは導入Guildも異なります。Bot「shovel_develop」は開発用Botのため、テストが不十分な状態で運転することもあります。そのため、重大な不具合発生の危険性があります。友人であっても大きな迷

惑をかけることはあってはならないので、開発者と試験対象Botしかいない開発用Guildで動作確認を行っています。

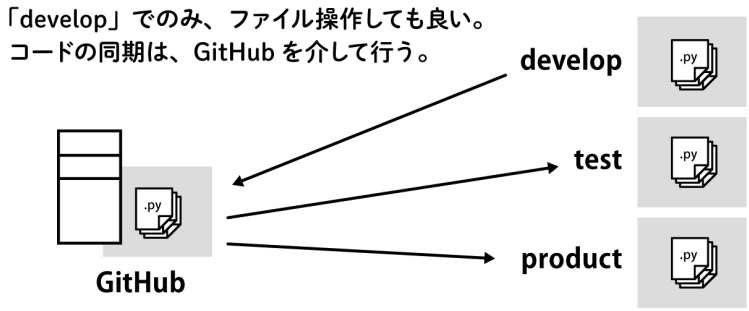
Bot「shovel_test」は、多少の不具合を許容できるGuild、具体的に言うと開発者が管理するごく親しい友人の常駐するGuild、友人の管理するGuildで試運転させてもらっています。

このようにBotを段階的にリリースすることで、致命的な不具合を起こして本番環境のユーザーの体験を損なう可能性を減らせます。

7.3.4　実装時のコード移行の流れ

図7.4をごらんください。まず、「develop」のコードを編集します。このとき、Bot「shovel_develop」で動作確認しながら開発します。ある程度開発が完了したら、「test」にコードを同期し、Bot「shovel_test」を稼働させて友人の協力を得ながら運用テストを実施します。運用テストで問題が発生した場合、また「develop」でのコード修正と動作確認をやり直します。運用テストで問題が発生しなければ、ようやくリリースです。「product」にコードを同期し、Bot「shovel」を稼働させます。各ディレクトリのコード同期は、GitHubのリポジトリーを通して行います。

図7.4: 実装時のコード移行の流れ



重要なのは、「test」や「product」で直接ファイル操作を行わないことです。あくまで、「develop」からコードを同期してくるようにします。これは、修正したらまずは動作確認、どんな簡単な変更でもぶっつけ本番でリリースをしないことを守るためのルールです。

また、GitHubのHookを利用した自動デプロイ等は行っていません。あくまでGitとGitHubはバージョン管理としてのツールとして利用し、デプロイやリリースのタイミングは別途管理するためです。

7.3.5　venvによるPython環境の分離

Pythonのパッケージ管理ツールであるpipは、デフォルトでOSのグローバル領域にモジュールをインストールします。このとき、「Bot1ではdiscord.pyのv0.xを動かしたいが、Bot2ではdiscord.pyのv1.xを動かしたい」等の要求が出てくると、手順が煩雑になります。そもそも、あるプロジェクトのみで使うモジュールをグローバル領域にインストールするというのに違和感もあります。

そこで、venvを使います。venvを使えば、プロジェクトごとにPythonインタープリターや外部パッケージのバージョンをすぐに切り替えられるようになります。プロジェクトごとに作成する仮想環境ディレクトリ内のみにインストールされるので、当然プロジェクト間でバージョンの衝突を起こすこともなくなります。

また、使いたいパッケージは直接OSコマンドのpipで指定するのではなく、requirements.txtに記述し、pip install -r requirements.txtでインストールします。requirements.txtはプロジェクトの一部としてリポジトリーに登録しておきます。

このとき、requirements.txtにはそれぞれのパッケージのバージョンも指定しておきましょう。パッケージは、未指定であればその時点での最新版がインストールされるので、思わぬアップデートによりBotの機能が壊れてしまうことあります。各パッケージの変更を常に把握し、細かく追従するつもりがないのであれば、バージョンは固定しておきましょう。こうしておくことで、リポジトリーさえあればどこでもすぐに同じ環境を再現できます。

7.3.6　DBとその中身のデータ

DBはそれぞれの環境ごとに用意します。中身のデータはそれぞれで異なりますので、DBMS内に環境ごとのDBを作成しておきましょう。また、DBに思わぬ変更が発生しないよう、DBMSのユーザーも環境ごとに作成しておくのが良いでしょう。

7.4　コード変更なしに環境を切り替える方法

7.4.1　環境切り替えの重要性

前の節では、同じBotでも、開発と本番でBotアカウントや参加Guildを分けるべきであるということについてお話ししました。また、DBも環境ごとに作成しました。でも各環境のshovelがそれぞれのBot、Guild、DBを切り替える仕組みはどのように実現すればいいのでしょうか？

コードを直接編集してパラメーターを変更するというのがもっとも単純明快ですが、これは良い方法だとはいえません。理由は2点あります。まず、手作業で編集しているとミスが発生します。次に、テスト用にコードを変更したつもりが、製品版をそのコードで動かしてしまうという誤りも発生し得ます。

そこで、環境ごとにコードをまるごと複製し、環境により動作を変更したい部分のみを設定ファイルでカスタマイズするという方法をとります。この方法はDiscord Botでないアプリケーションにも適用できますが、以下ではPythonおよびdiscord.pyを例として具体的に説明します。

7.4.2　フォルダー・ファイル構成

shovelのフォルダー構成をリスト7.1に示します。これは環境切り替えに関連するファイルを抜粋したものです。

リスト7.1: shovel開発 フォルダ構成

```
bot/
├── develop (開発用)
│   ├── .gitignore (1)
│   └── shovel/
│       ├── config.py (2)
│       └── config.py.sample (3)
├── test (テスト用)
│   └── developと同様
└── product (本番用)
    └── developと同様
```

それぞれのファイルの用途を見ていきましょう。

7.4.2.1　.gitignore

config.pyには、トークンやDBパスワードなどの非公開にすべき文字列が含まれます。これらはバージョン管理ツールのリポジトリーに登録すべきでない情報ですし、開発系・テスト系・本番系で共有しない情報です。そのため、リスト7.2のように、.gitignoreファイルにconfig.pyを記入しておきます。しかし、このままでは、config.pyというファイルが必要であることや、どのように書けばいいかの情報がリポジトリーに登録されません。そこで、config.py.sampleを用意します。

リスト7.2: .gitignore

```
config.py
```

7.4.2.2 config.py.sample

config.pyのテンプレートとなるファイルです。これをリポジトリに登録しておくことで、ファイルの項目を管理できます。項目の追加・削除があった場合、コードの変更と同時に、config.pyの項目の変更も履歴に残るので安心ですね。テンプレートはリスト7.3のように書きます。

リスト7.3: config.py.sample

```
DISCORD_TOKEN = "discord_bot_token"
DB_USER = "username"
DB_PASSWORD = "password"
DB_NAME = "table_name"
DB_HOST = "db_host"
DB_PORT = 00000
IS_DEBUG = False
```

7.4.2.3 config.py

それぞれのディレクトリにconfig.pyというファイルを配置します。これが環境切り替えのキモとなるファイルです。それぞれの環境用に個別のconfig.pyを作成することで、全環境で共通のコード本体を使いながら、パラメーターを環境ごとに切り替えられます。内容の設定例をリスト7.4、リスト7.5に示します。

リスト7.4: テスト用config.py サンプル

```
= これはテスト版の設定例です。
DISCORD_TOKEN = "test_discord_token.XXXXXXXXXXXXXXXXXXX"
DB_USER = "test_user"
DB_PASSWORD = "test_user_password"
DB_NAME = "test_db"
```

```
DB_HOST = "localhost"

DB_PORT = 12345

IS_DEBUG = True
```

リスト7.5: 本番用config.py サンプル

```
= これは製品版の設定例です。

DISCORD_TOKEN = "product_discord_token.XXXXXXXXXXXXXXXXXX"

DB_USER = "product_user"

DB_PASSWORD = "product_user_password"

DB_NAME = "product_db"

DB_HOST = "sample.com"

DB_PORT = 54321

IS_DEBUG = False
```

第8章　実装 - いざ、コーディング

　この章では、もっとも狭義での「プログラミング」であるコーディングについて説明します。リファクタリングによる技術的負債の返済、shovelを例にした実装上の工夫についてもお話します。

8.1　実装の流れ

8.1.1　shovelの場合

shovelは、図8.1のとおり、以下4つのステップを回すことで開発を行っています。

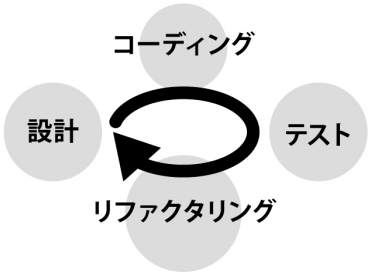
・設計

・コーディング

・テスト

・リファクタリング

図8.1: 4つのステップ



設計とはいえ、事前にクラス図や関数仕様書を書くといったような細かい内部設計はしていません。もちろん大きな機能を実装するときは、どのようなクラス分割にすればうまいつくりになるか、実装する前に考えることはあります。このときもそれぞれのクラスのメソッドひとつひとつについて考えることはせず、それぞれのクラスの責務とインターフェイスについてのみ検討します。

また、3つのタイミングでリファクタリングを行っています。まず、機能が完成したとき。次に、機能を追加するとき。最後に、技術的負債（後述します）が蓄積されたときです。

8.1.2　まずは動くものをつくろう

はじめてメッセージの読み上げに成功した読み上げBotは、たったひとつ、40行のPythonファイルにすべてが入っていました。クラスもなく、エラー処理もログもなく、変数の命名も適当な、ただ「メッセージを読み上げることができる」というだけのコードでした。機能追加に伴い、コード追加とリファクタリングを繰り返すことで、shovelは育ってきました。discord.ext.commandsフレームワークを使ったBotへとリファクタリングしたのは、はじめてメッセージの読み上げに成功したときから、約1年後のことでした。

綺麗なつくりにすることも保守性という観点でとても大切ですが、保守性を気にしてBotができあがらないようでは本末転倒です。まずはとにかく書きましょう。バージョン管理含む開発環境をしっかり整っていれば、どんどん書いてどんどん壊すことができます。shovelが19ファイル、延べライン数は4,000行という大きなプロジェクトになった今も、この考え方は変わりません。

8.2　リファクタリングで保守性を担保！

8.2.1　リファクタリングとは

リファクタリングとは、アプリケーションの動作自体はそのままに、コードをきれいにすることを指します。きれいにする、というと漠然としていますが、自分なりの「よいコード」にしていく作業だと思っていただければと思います。重要なのが「アプリケーションの動作自体はそのまま」という点です。コードの中身だけではなく、アプリケーションの振る舞いを変えてしまうのはリファクタリングとは呼べないことに注意してください。リファクタリングはあくまで、コードの構造を整理することを目的とした作業です。

8.2.2　リファクタリングの帽子

「実装の帽子をリファクタリングの帽子に被りなおす」という言葉を聞いたことがあるでしょうか。これは、「実装=ひたすら機能の完成や不具合の修正をめざしたコーディング」と、「リファクタリング=機能を損なうことなくコードを整理する作業」は同時に行ってはいけない、ということを示した格言です。まずは不格好なコードでも構わないので動くコードを完成させ、後からきれいに直すのがセオリーです。とはいえ、ボロボロなコードで大きなものを作ってしまうと後が大変なので、段階を追って実装とリファクタリングを繰り返すのがよいでしょう。たとえば、「ユーザーからメッセージを受け取ったら、その内容を加工して同じチャンネルに投稿する」という機能を実装したいときであれば、以下のようにします。

1. ユーザーからメッセージを受け取る処理を実装し、メッセージ内容をログに出してみる。
2. 1のコードをリファクタリングする。動作確認を行う。
3. ユーザーから受け取ったメッセージを、そのまま同じチャンネルに投稿する処理を実装する。
4. 3のコードをリファクタリングする。動作確認を行う。
5. メッセージの内容を加工する処理を追加する。
6. 5のコードをリファクタリングする。動作確認を行う。
7. すべてのコードをリファクタリングする。動作確認を行う。

今回の例のように、ひとつひとつの処理が単純である機能にここまでの詳細なサイクルは必要ありませんが、複雑な機能についても、考え方は同じです。書く、動かす、リファクタリングする、というサイクルを自然に回せるようにするとよいでしょう。

8.2.3　リファクタリングと自動テスト

アプリケーションの動作を変更しないことがリファクタリングの条件です。つまり、リファクタリングの前後でアプリケーションの振る舞いが変わっていないことを保証する仕組みをととのえる必要があります。リファクタリン

グを実施するプロジェクトでは同時に自動テストの仕組みが整備されていることが大半です。

しかし、Discord BotやWebアプリケーションでは、テストコードからコマンドの発火などのイベントを擬似的に起こすのは難しく、メッセージをテキストチャンネルに送信するといった仕様を単体テストで確かめることは難しいです。つまり、テストコードによる単体テストの自動化でカバーできる範囲が限られています。なので、リファクタリングを実施する際には丁寧な動作確認をするようにしましょう。

shovelではjishakuというライブラリーを利用し、テスター代理Botを使った自動回帰テストの仕組みを作っています。リファクタリング前後にこの自動回帰テストを実施することで、shovelの振る舞いに変化がないことを保証しています。jishakuによるテスト自動化については、第10章「Discord Botのテスト自動化」をごらんください。

8.2.4　よいコードの基準

よいコードとは何か、というテーマは奥が深く語りきれないものではありますが、自分なりに気をつけている詳細な観点をリストアップしてみました。コードについては、Dustin Boswell、Trevor Foucher、角征典『リーダブルコード』（2012　オライリー・ジャパン）という有名な書籍があります。これは業種や言語、趣味業務をとわず、ぜひ一度読んでいただきたい書籍です。ためになるだけでなく、ユーモアたっぷりで読み物としてとてもおもしろい本です。

8.2.4.1　読みやすいか

・変数名・関数名・クラス名は適切か
・命名に統一感はあるか
・モジュール、クラス、メソッドの大きさは適切か
・スマートさを求めるあまり可読性を失っていないか

8.2.4.2　修正しやすいか

・同じようなコードが分散して存在してはいないか
・共通処理を切り出せる部分はないか
・依存を適度に切り離したつくりになっているか

8.2.4.3　オブジェクト指向の原則に沿っているか

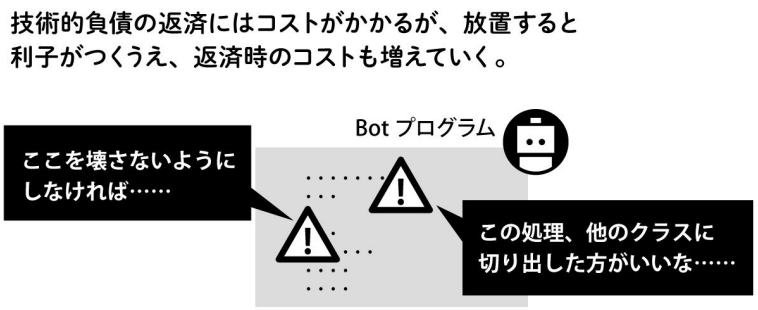
・各モジュール、クラスに不自然なインターフェイスはないか
・各モジュール、クラスのメンバーは責務にあっているか
・責務が多すぎるクラスは存在しないか

8.2.5　技術的負債はこまめに返済しよう

前項では、まずは動くコードを書き上げて、その後リファクタリングすればよいという話をしました。ここで、「とりあえず動くコード」をそのままにしておくと、何が起きるでしょうか？　この「とりあえず動くがあまり品質のよくないコード」のように先送りにした問題を指して、「技術的負債」と呼びます。コードにたまった技術的負債は、すこしの仕様変更をするだけのコード変更に多くの時間がかかったり、ある機能のためのコード修正をしたときまったく関係ないはずの機能に影響が出るなどの事態を招きます。

サービスを運営していると、機能を追加したり、不具合を修正することについ集中してしまいがちです。しかし、そのような技術的負債をそのまま放置した場合、コードを触る余分なコストがかかりつづけるだけにはとどまらず、そのコストは増えていきます。技術的負債が発生していることがわかったら、可及的速やかに解消すべきです（図8.2）。

図8.2: 技術的負債があるということ



技術的負債は、たとえ細やかにリファクタリングしていても、ふと気付くとコードに紛れ込んでいるものです。機能追加などのためにコードを触っていて、改善すべき実装や効率化できる処理に気づいたときは、コード中にコメントで内容をメモしておきましょう。もちろんコード中ではなくGitHubのissueや手元のTODOリストにメモしても構いません。すぐに手をつけたくなるかもしれませんが、途中の作業がある中でリファクタリングをするのは、変更管理の観点からもパフォーマンスの観点からも好ましくありません。機能追加や不具合修正などの合間、独立したタイミングで要改善部分の修正を行うとよいでしょう。

8.3　保守性を高める実装

8.3.1　Databaseモジュールの存在意義

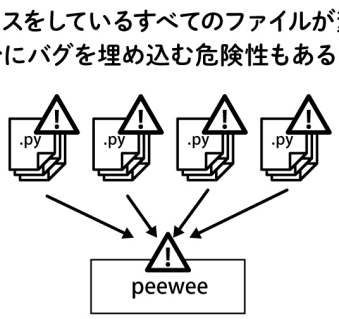
第4章「shovelのソフトウェア構成」では、「Database」というモジュールがあることをお話しました。繰り返しになりますが、機能を満たすだけであれば、各モジュールで直接peeweeを操作してもまったく問題はありません。それどころか、わざわざこのモジュールを通すのは、コーディングの量も動作する際の処理の量も増え、煩雑になることを意味します。なぜこのようなつくりになっているのでしょうか。ここで、データへのアクセス方法が変わるときのことを考えてみましょう。たとえば以下のようなときです。

・peeweeではなくSQLAlchemyに乗り換えたくなったとき
・ORMの利用をとりやめ、独自にDBへアクセスする処理を書きたくなったとき
・DBではなく、まったく違うデータソースとのやりとりに変更したくなったとき

8.3.1.1　Databaseモジュールがないとき

このようなとき、各モジュールにpeeweeの利用を前提としたコードが散らばっていたらどうなるでしょうか？そのすべてを洗い出し、それぞれについて新しい方式のものに修正を行い、さらには各モジュールの他の部分に影響が出ていないかをひとつひとつ確認する必要が出てくるでしょう。その検証はすべてを作り直したときと同じような、大変な作業になります（図8.3）。

図8.3: Databaseモジュールがないとき

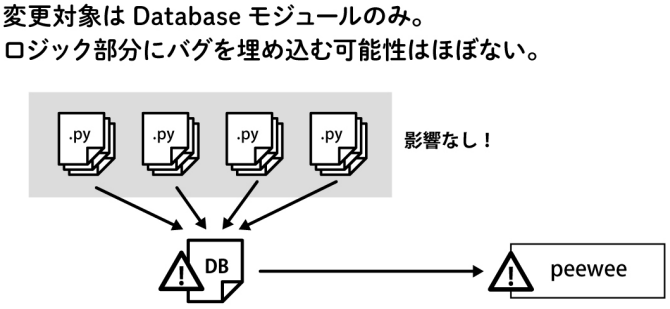


8.3.1.2　Databaseモジュールがあるとき

データアクセスがひとつのモジュールに統一されている場合はどうでしょうか。まず、モジュール「Database」以外を触る必要は一切ありません。変更の前後でインターフェイスが変わらないようにだけ気をつけていればいいので、コードの変更も難しくありません。そしてテストですが、もちろんデータへのアクセス方法を変更した

ことによる影響が出ていないか全体を確認することは必要です。しかし、ロジックの誤りを埋め込んだり、まったく関係のないコードにバグを埋め込んだりする可能性は少ないと言えるでしょう（図8.4）。

図8.4: Databaseモジュールがあるとき



データにアクセスするためのモジュールを用意することで、データアクセスの方式が柔軟になり、保守性が高い設計となっていることがおわかりいただけたでしょうか。このように、実装・処理にオーバーヘッドをかけても、依存関係を減らすことには、非常に大きな価値があるのです。

8.4　ユーザビリティのための実装

8.4.1　モジュールの動的リロードの実装例

第6章「ユーザーの使用を中断しない更新」にてモジュールのリロードによるアップデートについて触れました。ここでは、モジュールのリロードによるアップデートを実現するためのshovelの実装について説明します。

Pythonの標準モジュール「importlib」には、reloadという関数があります。これは以前にインポートされたモジュールをリロードするという処理です。モジュールのリロードは、開発者専用cogAdminに含まれるコマンドとして実装しています。内容をリスト8.1に示します。

リスト8.1: reload_module コード

```
@commands.command()
async def reload_module(self, ctx, module_name=None):
    if not module_name:
        await ctx.send("モジュール名が必要です。")
        return
    try:
        import importlib
        module = importlib.import_module(module_name)
        importlib.reload(module)
    except (ModuleNotFoundError,
            ImportError) as e:
        await ctx.send("モジュールのリロードに失敗しました。")
        await ctx.send(e)
    else:
        await ctx.send("モジュールのリロードに成功しました。")
```

8.4.1.1　動的リロードを意識したモジュール実装

動的リロードの対象となるモジュールを実装する際にも注意が必要です。グローバル変数の初期化は、グローバル変数が定義されていないことを確認のうえで行う必要があります。すでに一度初期化を終え、必要なデータが入っているグローバル変数が、リロード時に初期値で上書きされないようにするためです。また、from <module> import <name>形式でインポートされた関数やクラスは、reload後も以前のコードのままとなります。リロードされることを想定する場合、import <module>でモジュールごとインポートし、<module>.<name>の形式でインポートするようにしましょう。例をリスト8.2に示します。

リスト8.2: reloadを想定したモジュールの処理

```
= 悪い例
from mymodule import mymethod
mymethod()


words = {}


= 良い例
import mymodule
mymodule.mymethod()


try:
    words
except NameError:
    words = {}
```

Discord Botにおけるcogのリロードは、jishakuを導入すればかんたんに行えます。Discord上でjishaku reload mycogコマンドを実行すると、mycogはリロードされます。

第9章　テスト - コードの品質をまもる、最後の砦

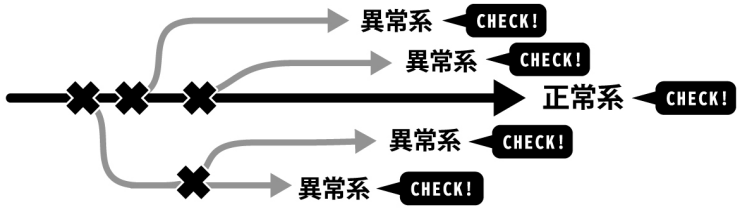
　この章では、テストについてお話します。ソフトウェアのテストは非常に奥が深いものであり、本書では筆者の方法と考え方についてを簡単に説明します。また、テストに関する用語（単体テスト、結合テストなど）は業界や組織によって方言があることや、本書の内容が普遍的に正しいわけではない点をご了承ください。

9.1　テストとは何なのか

9.1.1　Botの動作をテストで確認しよう

「親しい友人向けにBotを運営していて、使ってるうちにおかしい動作が見つかったので直す」という経験は、Bot開発者なら誰にでもあるのではないかと思います。実はこれは立派なテストです。また、友人に使ってもらうまえに、自分用のGuildを作り、そこでBotをためしに動かしてみたこともあるでしょう。これもテストといえます。テストというのは堅苦しいものでも、大げさなものでもなく、「Botが思ったとおりに動くことを確かめる」という目的で行われるすべてのことを指している言葉なのです。第6章「品質を上げるための設計のポイント」にて、正常系だけでなく異常系も漏れなくあぶり出し対応を検討する必要があると述べましたが、テストというのは、正常系、異常系、すべての動作が仕様通りになっていることを確認するものです（図9.1）。

図9.1: すべてのパターンの動作が仕様通りであることを確認する



9.1.2　テストの種類

ひとくちにテストといっても、さまざまなものがあります。もっとも小さなテストは、プログラムの関数、メソッドが仕様通りに実装されていることを確認するもので、単体テストとよばれます。実際にアプリケーションを実運用と同じように操作し、仕様書どおりに動くことを確認するテストもあり、これはソフトウェア結合テストと呼ばれます。shovelでは、「機能のテスト」「回帰テスト」「実稼働によるテスト」という3つのテストを実施しています。

9.1.2.1　機能のテスト

機能追加時、機能改修時に実施するテストです。そのとき触った機能と、関連する機能について、正常系・異常系すべてを網羅するテストをします。実装がすべて終わってからテストするというよりは、実装とテストを繰り返して少しずつ機能を作りあげ、最後にすべてのテストを再度実施するという手順で開発を行っています。

この「機能のテスト」については、次節で詳細な設計・実施手順を紹介します。

9.1.2.2　回帰テスト

機能追加の実装とテストを完了し、いよいよ新バージョンのBotをリリースしようというときに大切なのが、触っていない機能をふくめた全体のテストです。「触っていないのだから何もかわっていない、テストなど不要ではないか」と思われるかもしれませんが、同じプロセスに存在するコードを触った以上、絶対に影響がないとは、まず言い切れるものではありません。新機能開発中はその機能に関連する部分を重点的に動かし、その他をあまり動かさないケースが多く、新機能と直接関係のない箇所に不具合が生じていた場合に発見が遅れます。また、利用している外部パッケージのアップデートや、利用している外部APIの仕様変更などにより不具合が発生することもあります。ですので、「Bot全体の機能が正しく動いていること」を網羅的に確認することは重要です。

回帰テストは、基本的には「機能に対するテスト」で実施したテスト項目を再利用します。ただし、機能を実装したときと同じような細かいテストは必要ない場合もあります。「機能に対するテスト」のテスト項目のうち、重要な部分のみピックアップして実施すればよいでしょう。

9.1.2.3　実稼働によるテスト

これをテストと呼ぶかどうかは議論がありますが、実運用もテストの一環として見ることもできます。実際に運用を開始して多くの人の手で使ってもらい、不具合が出ないかを確認します。また、一部のユーザーを対象にして本リリース前に新規機能を先行提供し、見つかった不具合を修正したり、反応をみて全体へ適用するかどうか検討するといった評価を行うこともあります。

9.2　テストを設計しよう

9.2.1　テスト設計を行うことのメリット

さきほど述べたように、「なんとなく操作して反応を確認する」「友人に使ってもらう」というのもテストですが、これは設計されていないテストです。これでも不具合を見つけることはできます。それでは、わざわざテストを設計することのメリットはなんでしょうか。

まず、適切に設計されたテストを実施することで、機能について網羅したテストを行えるので、安心してBotをリリースできます。テストを設計・実施すれば不具合が一切なくなるかというと、そうではありません。しかし、「少なくともこういう条件でこの機能に不具合が起きることはない」「確認すべきことはすべて確認した」という安心を得られます。ユーザーから不具合報告が上がっても、それがテストしたはずの内容であれば、冷静に対応できます。設計されていない曖昧な動作確認を実施しただけでは、すべてを確認したという保証はできないので、アプリケーションの品質にいまいち自信が持てないままになってしまいます。

また、テスト設計をすることで、仕様の設計漏れに気づくこともあります。仕様の設計をする際には、どうしても実現したい機能そのものに目が行き、異常系を捕捉しそこねてしまいがちです。テスト設計は不具合が起きそうな部分を探すという意地悪な目線で検討するので、仕様設計時に見逃したものにも気づきやすいのです。

最後に、テストを設計し手順を整理しておくと、テストフェーズを自分ではない誰かに任せられます。この「誰か」にはあなた以外の人間はもちろん、テスト用Botやアプリケーションなどのプログラムを含みます。

9.2.2　テスト項目を決める

仕様を決めることは、テスト項目を決めることに等しいといえます。ですので、shovelでは新しい機能を追加するとき、まずはテスト設計をすることからはじめます。

まずは、どのようなことを確認したいかをリストアップします。たとえば、「複数行を読み上げるかどうかの設定を追加し、設定がONであれば2行目以降を読み上げる」という機能をBotに追加するときのことを考えましょう。確認すべき内容は2点です。

・複数行を読み上げるかどうかの設定ができること

・複数行読み上げ設定がONのとき、2行目以降も読み上げること

文章にすると簡単ですが、これらが本当に意図通りに実装されているかを確認するには、もう少し細かい手順が必要です。

9.2.3　テスト項目の細分化

テスト項目を挙げたら、確実に確認するためにもっと細かい項目を考えていきます。前項に挙げた例でいうと、「複数行を読み上げるかどうかの設定ができる」という項目には、「設定のON、OFFを自由に設定できる」という意味が込められています。そこで、それを確認できるよう、項目を細分化します。

・複数行を読み上げるかどうかの設定ができること
- 複数行読み上げ設定をONにできること
- 複数行読み上げ設定をONにしたあと、OFFにできること
- 複数行読み上げ設定をON→OFF→ONにできること
- 複数行読み上げ設定に異常な値を渡すと、エラーになること

同様に、「複数行読み上げ設定がONのとき、2行目以降も読み上げる」についても細分化します。この仕様の行間には、「複数行読み上げ設定がOFFのときは、2行目以降は読み上げない」という仕様が隠れています。これをテストで明確に確かめておかないと、後々「どんな設定でも複数行読んでしまう」という不具合が発生しているとわかったときに、いつから発生していた不具合なのかがわからなくなってしまいます。また、2行目以降を読み上げるかそうでないかという処理を追加することで、1行の読み上げにも影響が出ることも考えられます。そうでないことを保証するため、単一行の読み上げについてもテストをしておきます。

・複数行読み上げ設定がONのとき、複数行投稿の2行目以降も読み上げること
- 複数行読み上げ機能がOFFのとき、複数行投稿の2行目以降は読み上げないこと
- 複数行読み上げ機能がONのとき、1行のみの投稿を正常に読み上げること
- 複数行読み上げ機能がOFFのとき、1行のみの投稿を正常に読み上げること

このように、テスト設計は、仕様に込められた暗黙の処理の実装を決めることと裏表の関係です。そのため、仕様設計とテスト設計は平行して相互補完的に行っていくと、どちらかいっぽうだけに漏れがないか頭を悩ませるより効率がよいといえます。テスト項目の設計については、第6章の異常系に関する記述も参考にしてください。

9.2.4　テスト手順の記述

次にテスト手順を記述します。これは、Botのような小規模なアプリケーションならそこまで神経質にすべて記述する必要はありませんし、テスト実施者が自分しかいないなら、項目を見れば何をするのかわかります。ですが、テスト項目を見ただけでは何をするべきかわからないような複雑な手順である場合や、テスト実施者が開発者以外である場合は、テスト手順を記録しておくようにしましょう。

手順には、それぞれの手順を実行したあとの確認項目もあわせて記入しましょう。手順の実施後に確認することがない場合、「なし」としてもかまいません。また、ひとつのテスト項目に対してひとつの手順を用意する必要もありません。たとえば、前項で「複数行を読み上げるかどうかを設定できること」にはテスト項目が4つありましたが、このテストはすべて、図9.2の手順でまとめて行えます。

図9.2: テスト手順の例

No.	手順	確認項目
1	!sh read_multi on を実行する	正常に変更が完了した旨のメッセージが表示されること
2	!sh server_settings を実行する	「複数行読み上げ設定」 が ON になっていること
3	!sh read_multi off を実行する	正常に変更が完了した旨のメッセージが表示されること
4	!sh server_settings を実行する	「複数行読み上げ設定」 が OFF になっていること
5	!sh read_multi on を実行する	正常に変更が完了した旨のメッセージが表示されること
6	!sh server_settings を実行する	「複数行読み上げ設定」 が ON になっていること
7	!sh read_multi ??? を実行する	異常値により変更できない旨のメッセージが表示されること

ここで重要なのが、あくまで「テスト項目の検討」と「テスト手順の作成」は分けて行うべきであるということです。前者では、「何をテストすべきか」を考えます。後者では、「テスト項目のテストを実施するために必要な手順はなにか」を考えます。テスト項目を考えるとき、テスト手順が同時に浮かんで来ることもありますが、あくまで別のものとして分けて書いておきましょう。これをひとまとめにしてしまうと、「ひととおりの操作をした。だからすべてのテストができた」と考えてしまいがちです。

次に、手順とテスト項目の対応を明文化しておくことも大切です。上記のように、テスト手順を表にしているなら、各テスト項目に連番を振り、「テスト項目No.」の列を設けてそこに記入しておくとよいでしょう。

9.3　テストを実施しよう

9.3.1　試験を実施し、記録を残す

設計した手順どおりにテストを実施します。テストを実施したら、実施した日付とバージョン番号（ない場合、Gitのrevisionでも可）、試験結果（合格・不合格）を記録しておきます。それぞれのテスト項目について、結果をひとつひとつ記録しなくても構いません。すべてのテストが合格となったことをまとめて記録しておいてもOKです。ひとつひとつの結果を記録したい場合、テスト手順の表をコピーして結果を書き入れていけばよいでしょう。

万が一、テスト実施中にテスト設計自体の誤りを見つけた場合は、面倒でもテスト実施を中断し、テスト仕様の誤りを修正してから続行するようにしましょう。もし修正を後回しにして、そのまま忘れてしまった場合、実際のテスト内容と記録上のテスト内容に食い違いが発生します。これでは、もはやテストを行った意味がありません。誤字程度の単純な誤りでも、かならず修正してからテストを実施しましょう。

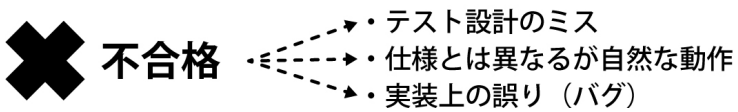
9.3.2　エビデンスを残す

テストを実施したら、エビデンス（証拠）を残しておくのが通例です。自分が実施したテストにもエビデンスは必要です。なぜなら、「合格」としたテスト項目に関連する不具合が出たとき、「本当にそのとき合格であったか」を確認する手段になるからです。テスト結果のエビデンスがない場合、実際は不合格のはずなのに見間違いで合格にしてしまった項目があるかもしれないと思っても、あとで特定できません。不具合箇所を特定するにはテストをやりなおさねばならないのです。そのような事態を避けるためにも、テスト結果のエビデンスは残すようにします。Discord Botのテストであれば、テスト用Guildのログがそのままエビデンスになります。

9.3.3　テストが「不合格」となったら

テストを実施していると、テストが「不合格」、つまり意図しない動作になっていることがあります。このとき注意したいのが、「テスト不合格」＝「バグ、不具合」ではない、という点です。テストに不合格なのであれば不具合じゃないかと思われるかもしれません。もちろん大半は実装の誤りによるバグであることが多いです。しかし、テスト設計に誤りがある可能性や、テスト設計で想定している仕様より実際の動作のほうがユーザーにとって自然でわかりやすい可能性もあります。（図9.3）。

図9.3: テストの不合格＝バグではない



　そのため、テストが「不合格」となったら、その不合格をどのように取り扱うかを検討しなくてはいけません。テストを修正するのか、コードを修正するのか、仕様を再検討するのか、柔軟に検討しましょう。もちろん、修正後は再度テストを行います。すべてのテスト項目が「合格」になるまでテストを繰り返しましょう。

第10章　Discord Botのテスト自動化

　この章では、テストを自動化することの意味と、Discord Botのテストを自動化する方法を紹介します。また、テストの自動化に使っているライブラリー「jishaku」についても紹介します。

10.1　テスト自動化とは

10.1.1　なぜ自動化するのか？

テスト自動化をするモチベーションは多岐にわたります。筆者がshovelのテストを自動化するうえで、感じているメリットについて記します。

10.1.1.1　手順の漏れと誤りを防げる

テストを手作業で行っていると、手順に抜けが生じたり、見間違いで合否判定を誤ることがあります。テストの実施を自動化することで、このようなヒューマンエラーの起きる余地がなくなります。

10.1.1.2　リファクタリングがしやすくなる

リファクタリングとは、「アプリケーションの振る舞いを変えることなく内部構造を変更する」ことです。リファクタリングの条件、「アプリケーションの振る舞いを変えることなく」を保証するためにはテストが必要です。自動化によりテストの実施にかかる手間が減ることで、リファクタリングに取り組みやすくなります。

10.1.1.3　どのようなテストをしたかがわかりやすくなる

テストを自動化する際には、自動化用のデータ（テストコード含む）を用意します。このデータには実施したテストの手順と合否判定のロジックが含まれますので、テストを実施してから時間がたってもどのようなテストをしたかが一目瞭然です。ただしテストの意図まではわからないので、テスト仕様書を作るか、テストコード内のコメントなどで補足する必要があります。

10.1.2　何を自動化するのか？

自動化できる対象は2点あります。まずは「機能のテスト」です。アプリケーションを実装する際は、設計したテストに基づき、テストコードを書きます。ここで書いたテストにすべて合格すれば、機能の開発が終了したといえます。次に、「回帰テスト」も自動化できます。機能のテストに用いたテストコードは、そのまま回帰テストに流用できます。しかし、テストによっては時間経過を必要とするものもあり、そのようなテストについては回帰テストからは省いてもよいでしょう。

10.1.3　どうやって自動化するのか？

discord.pyのコマンド処理には、discord.Messageやdiscord.Contextといったデータクラスが密接に絡んできます。これらのデータクラスの内容を齟齬なく作成するのは困難です。筆者はshovelのUIテストを自動化したいと考えたとき、既存のユニットテストフレームワークをそのまま適用するのは難しいと感じました。

しかし、機能が増えるにつれ、回帰テストを行うのにかかる時間も増えていき、どうしてもUIテストの自動化をしたいと考えるようになりました。調査の結果、既存のフレームワークに頼るのは諦め、自分でテスターBotを作って自動化することにしました。

筆者がここで紹介するのは、いわゆるテストクラスというのではなく、単なるBotプログラムです。テスターBotが行えるのは、コマンドを発行すること、そのコマンドに対するshovelの応答結果をチェックすることです。次節からは、このテスターBotの実装方法について説明します。

10.2　Botのテストに欠かせないモジュール「jishaku」

10.2.1　jishakuとは

jishakuは、discord.ext.commandsフレームワークで作られたDiscord Botをデバッグ・テストするためのライブラリーです。MITライセンスでリリースされています。

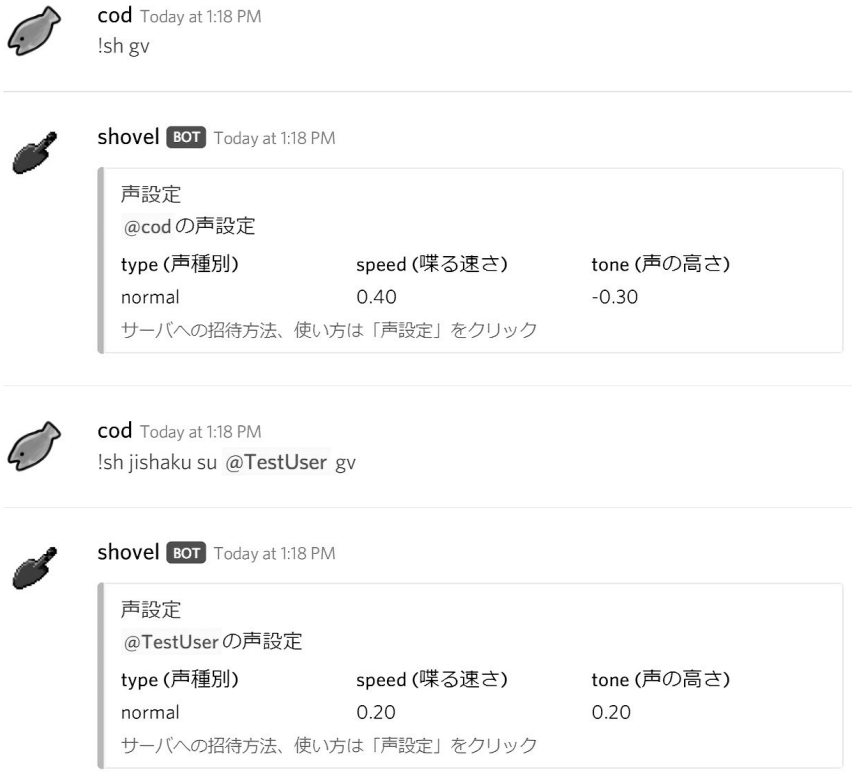
10.2.2　jishakuにできること

jishakuを組み込むと、具体的にどのようなことができるのでしょうか。jishakuをBotに導入すると、jishaku <jishakuコマンド>という形式で、さまざまなjishakuコマンドを使うことができます。jishakuコマンドを使うと柔軟にBotをデバッグできるため、デフォルトでBotのオーナーしか利用できないようになっています。jishakuコマンドの使用方法の一部を紹介します。

10.2.2.1　柔軟なコマンド実行

jishaku su <member> <command>コマンドは、memberで指定したユーザーがcommandで指定したコマンドを実行したとして、Botに処理を実行させるものです。具体例を見るのがわかりやすいでしょう。図10.1をごらんください。

図10.1: jishaku suコマンド 使用例



shovelにおけるgvコマンドは、コマンド実行ユーザーの声設定を表示するコマンドです。前半は、通常通りgvコマンドを実行した様子です。発言ユーザー「cod」の声設定を表示しています。後半では、発言ユーザーは前半と同じ「cod」であるにもかかわらず、shovelは「shovel_green」の声設定を表示しています。

このsuコマンドを使用できることが、jishakuを自動テストに用いる最大のポイントです。大半の自動テストでは、Botがそのままコマンドを発行すればよく、jishakuを使わなくてもかまいません。しかし、それではテストできないものもあります。たとえば、Guildの管理者権限を持つユーザー（以下、管理者ユーザーと呼ぶ）しか実行できないコマンドについて「管理者ユーザーがコマンドを実行しようとしたとき、実行できること」「一般ユーザーがコマンドを実行しようとしたとき、実行できないこと」を確認したいとします。このとき通常であれば、「管理者ユーザー」と「一般ユーザー」の両方からコマンドを実行する必要があります。しかしjishaku su <一般ユーザー>、jishaku su <管理者ユーザー>のようにjishaku suコマンドを使ってコマンド実行ユーザーを切り替えることで、単一のテスターBotからのコマンド発行で両方のテストが行えるのです。

このほかにも、指定したchannelでコマンドcommandを実行するjishaku in <channel> <command>コマンドもあります。

10.2.2.2 Pythonコードの実行

jishaku python <argument>コマンドを使用すれば、BotにPythonコードを実行させることができます。コード中では、_bot、_ctx、_message等の特別なコンテキスト変数を使うことができます。例として、リスト10.1をごらんください。

リスト10.1: Pythonコード実行の例

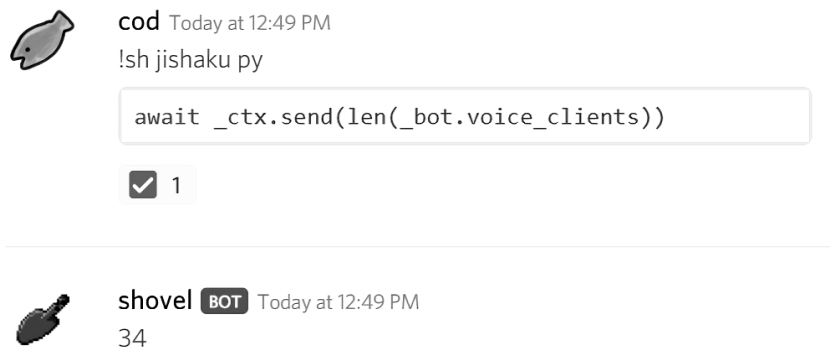
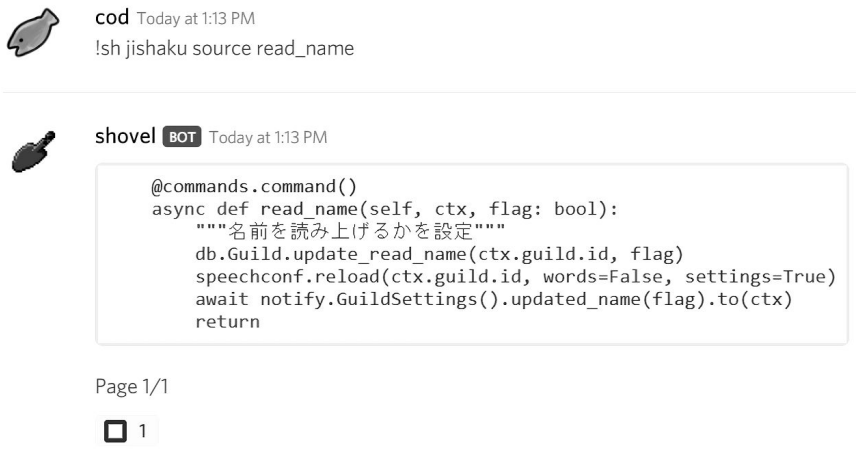
```
!sh jishaku py ```py
await _ctx.send(len(_bot.voice_clients))
```
```

これは、shovelのjishakuコマンドを使って、shovelに対して現在のボイスチャンネル接続数を出力させる命令のサンプルです。実行すると、図10.2のようになります。平日昼間のため、かなり接続数は少なめです。

図10.2: jishaku pyコマンド 使用例




#### 10.2.2.3　コードの閲覧

jishaku cat <filename>コマンドを仕様すれば、filenameで指定したファイルのソースコードを閲覧できます。シンタックスハイライト（ソースコードが見やすくなるよう色などをつけて表示すること）はファイルの拡張子や内容から自動で設定されます。また、ファイルの内容が長くなる場合、自動的にページ分割されて表示されます。jishaku cat <filename>#L10-20のように、ファイルの一部を行指定して読み込むこともできます。

Botコマンドの処理内容を見たい場合、もっと便利な方法があります。jishaku source <command>コマンドを使用すれば、commandで指定したコマンドのコードを閲覧できます。図10.3は、jishaku sourceを使い、shovelのコマンドread_nameのコードを閲覧する例です。
```

図10.3: jishaku sourceコマンド 使用例



10.2.3　jishakuの導入

10.2.3.1　インストール

　jishakuパッケージはPyPIに登録されているため、pipコマンドでパッケージ名を指定することで簡単にインストールできます。また、GitHubリポジトリーを指定して、開発版をインストールすることもできます。リスト10.2をごらんください。

リスト10.2: jishakuのインストール

```
安定版のインストール
$ python3 -m pip install -U jishaku<Enter>
開発版のインストール
$ python3 -m pip install -U git+https://github.com/Gorialis/jis
haku@master#egg=jishaku<Enter>
```

10.2.3.2　Botへの組み込み

　jishakuをBotに追加するには、Botインスタンスに対してjishaku cogを追加します。リスト10.3をごらんください。

リスト10.3: jishakuをBotに追加するサンプルコード

```
from discord.ext import commands
bot = commands.Bot(command_prefix="$")
bot.load_extension("jishaku") # Botへjishaku cogを追加している
```

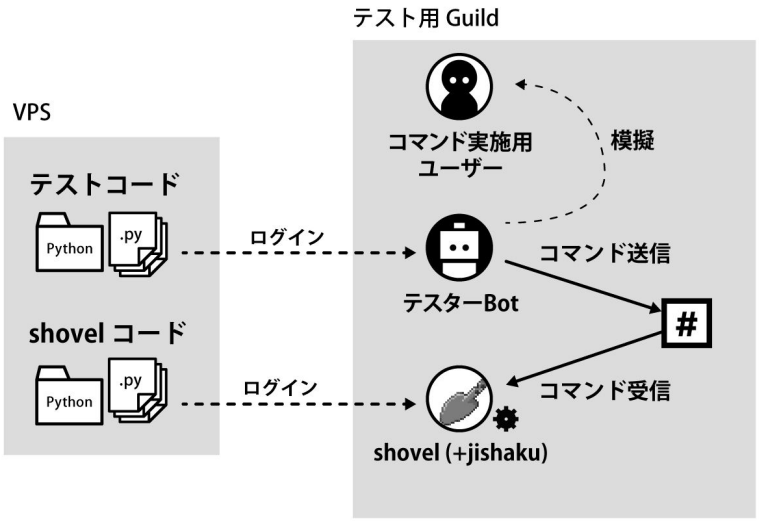
```
...
bot.run("<Discord Botトークン>")
```

10.3　テスト自動化の実装方法を紹介！

10.3.1　構成

ここからは、jishakuとテスターBotを使ったテスト自動化の仕組みについてお話します。テスト自動化の全体図を図10.4に示します。ここに出てくる「テスターBot」は、shovelや試験用Bot、開発用Botとは完全に別のBotです。テスターBot用のBotアカウントを取得しておいてください。

図10.4: テスト自動化の構成要素



10.3.2　自動テストのためにshovelを修正するポイント

テスターBotから自動テストを行えるようにするため、shovelにも少し手を入れる必要があります。

10.3.2.1　設定ファイルにテスターBotのIDを追加

自動テストに向けて、テスターBotを特別扱いする処理を組み込みます。その準備段階として、まず設定ファイルにテスターBotのIDを追加しましょう（リスト10.4）。

リスト10.4: config.py 追加内容

```
...
TEST_BOT_ID = <テスターBotのID>
```

10.3.2.2　テスターBotからのコマンド受付を追加

discord.ext.commandsフレームワークでは、Botから送られたコマンドは無視するようになっています。この制限を取り除くため、リスト10.5のようにコードを追加します。

リスト10.5: テスターBotからコマンドを実行できるようにするコード

```
class Shovel(commands.Bot):

    ...

    async def process_commands(self, message): ←①

        if message.author.id == config.TEST_BOT_ID: ←②

            ctx = await self.get_context(message)

            await self.invoke(ctx)

        else:

            await super().process_commands(message) ←③
```

1. コマンドの実行処理を行うprocess_commandsをオーバーライドします。
2. メッセージ送信者のIDが、config.TEST_BOT_IDと一致するか確認します。
一致する場合、続く2行で、コマンドを実行する処理を実施します。
3. 一致しない場合、通常のprocess_commandsに処理させます。

10.3.2.3　テスターBotからjishakuを実行できるようにする

jishakuコマンドはBotのオーナー以外には実行できません。なので、テスターBotをオーナーとしてみなすようコードを追加します。リスト10.6をごらんください。

リスト10.6: テスターBotからjishakuコマンドを実行できるようにするコード

```
class Shovel(commands.Bot):

    ...

    async def is_owner(self, user: discord.User): ←①

        if user.id == config.TEST_BOT_ID: ←②

            return True

        else:

            return await super().is_owner(user) ←③
```

1. Botオーナー判定処理を行うis_ownerをオーバーライドします。
2. 判定対象ユーザーのIDが、config.TEST_BOT_IDと一致するか確認します。
一致する場合、Trueを返します。

3. 一致しない場合、通常のis_ownerに処理させます。

10.3.3　テスターBotのコード

次はテスターBotを見ていきます。

10.3.3.1　config.py

テスターBotのパラメーターを管理するファイルです。トークンを含むので、Gitの管理対象からは外しておきましょう。リスト10.7をごらんください。TARGET_PREFIX、TARGET_IDには、テストしたい対象のプレフィックスとIDをそれぞれ記述します。このパラメーターがあることでテスト対象のshovel（本番用shovel、開発用shovelなど）を切り替えることができます。

リスト10.7: config.py

```
TOKEN = "<テスターBotのトークン>"
PREFIX = "<テスターBotのプレフィックス>"
TARGET_PREFIX = "<テスト対象Botのプレフィックス>"
TARGET_ID = <テスト対象BotのID>
```

10.3.3.2　tester_bot.py

テスターBotのソースコードの内容です。リスト10.8をごらんください。コマンド発行に用いる値と、コマンド実行結果の判定に使う値をリストに入れ、順次テストを実行しています。

リスト10.8: tester_bot.py

```
from discord.ext import commands
import discord
import asyncio
import config

OK_COLOR = discord.Color.blue()
NG_COLOR = discord.Color.red()

OK_EMOJI = '\N{WHITE HEAVY CHECK MARK}'
NG_EMOJI = '\N{CROSS MARK}'

bot = commands.Bot(command_prefix=config.PREFIX)
```

```
@bot.command()
async def read_limit(ctx):
    def check(m):
        return m.author.id == config.TARGET_ID and m.channel == ctx.channel

    params = (
        (4, NG_COLOR),
        (5, OK_COLOR),
    )

    for param in params:
        await ctx.send(f"{config.TARGET_PREFIX}jsk su TestUser "
                       f"read_limit {param[0]}")
        try:
            msg = await bot.wait_for('message', check=check, timeout=3)
        except asyncio.TimeoutError:
            await ctx.send(NG_EMOJI)
            continue
        result = (msg.embeds and msg.embeds[0].color == param[1])
        await msg.add_reaction(OK_EMOJI if result else NG_EMOJI)

bot.run(config.TOKEN)
```

10.3.4　動作イメージ（スクリーンショット）

まずテスターBotを実行します。

リスト10.9: テスターBotの実行

```
$ python tester_bot.py<Enter>
```

テスターBotとテスト対象BotがいるGuildで、!test read_limitを実行すると、自動でテストが実行され、合格となった場合チェックマークリアクションが付きます。図10.5をごらんください。ここでは、テスターBotは「soil」という名前で登場しています。

図10.5: テスターBot動作結果

10.3.5　さらに実用化するためのアイデア

以上の説明では、テストの判定結果を表示するために、「コマンドを発行されたshovelが応答した、そのメッセージにリアクションをつける」という方法を示しました。しかしこの方法だと、テスト項目数が多くなったとき、すべてのテスト結果を確認するのは大変です。実際のshovelのテストにおいてはこれまでに書いたものを発展させ、さまざまな工夫をしています。スペースの都合上ここではすべてを紹介できませんが、限られた時間で効率的にテストを実施するための主な施策を以下に紹介します。

・テスト終了時、実施したテストの数、合格したテストの数を表示する
・不合格となったテストの数と関数名を表示する
・テスト結果のうちひとつでも不合格のものがあれば、警告を表示する
・discord.ext.taskを利用し、定期的に自動で回帰テストを実行する
・回帰テストの結果をWebhookで通知する

第11章　アップデートのための作業

　実装とテストを終えたら、いよいよサービスのアップデートです。この章では、アップデートをより安全に実施するための「アップデートリハーサル」について述べます。また、shovelの大規模アップデートの際に行っている、ダウンタイムを短くするための工夫についても説明します。

11.1　アップデートリハーサルをしよう

11.1.1　アップデートの副作用

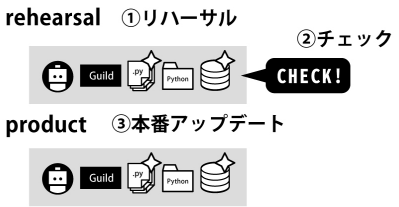
そもそもアップデートとは、バグ修正や機能追加など、ユーザーによりよいサービスを提供するために行われるものですので、ユーザーにとっては喜ばしいことであるはずです。しかし、アップデートに時間がかかりすぎてユーザーがサービスを使えない時間が多く発生したり、アップデートの過程でユーザーのデータを壊してしまったりしては元も子もありません。

アップデートを実施する際には、ユーザーのデータを損なわないことや、サービスのダウンタイムを最小限に抑えることが重要です。このとき有効なのが、アップデートリハーサルです。

11.1.2　アップデートリハーサルの意義

アップデートリハーサル（以下、リハーサル）の目的は、アップデートにかかわる作業手順を検証することです。作業手順の検証には2段階あります。1段階目は、まずBotの稼働を止めてDB構造を変更しその後再びBotを動かす、といった「アップデート手順書」を作成すること。2段階目は、前段階で作成した「アップデート手順書」のとおりに作業をすれば、アップデートが正しくスムーズに完了するのを確認することです。これらの作業は、「本番環境」をできる限り模した「リハーサル環境」で検証します。

図11.1: 本番アップデートの前に、リハーサル環境で検証する



いきなり本番環境でアップデートを実施せず、リハーサル環境をつくる理由は、トラブルの可能性をすこしでも減らし、サービスのダウン回数と時間を減らすためです。リハーサルは、本番同様のデータをわざわざ用意しなければならない点、二度手間となる点で、面倒なものではあります。しかしリハーサルを省くと、思わぬトラブルでサービスが起動できなくなったり、DBに置かれたユーザーデータを損なってしまったりといった重大な事態に繋がります。ユーザーの利便性向上のために行うはずのアップデートでユーザーに不便を強いては本末転倒です。ミスや誤りの可能性があるアップデート作業をするときは、面倒でもリハーサルをしっかり行いましょう。

11.1.2.1　テストとリハーサルの違い

テストはあくまで「サービス本体の機能のテスト」です。機能が意図通りに実装されているか、異常系に対して網羅した対応ができているかを確認するのがテストです。リハーサルは、データの移行手順やアップデートの順番など、「アップデート手順の検証」が目的です。

11.1.3　アップデートリハーサルをやってみよう

ここでは、アップデートリハーサルの手順を紹介します。

11.1.3.1　アップデート手順書作成

DB構造を変更した場合など、コードを更新する以外にアップデートのための手順が必要である場合、アップデートの手順書を作成します。リスト11.1は、実際にshovelのアップデート時に使用したアップデート手順書です。便宜上「手順書」と呼んでいますが、これはシェルスクリプトなどにしてしまってもかまいません。

リスト11.1: アップデート手順書 例

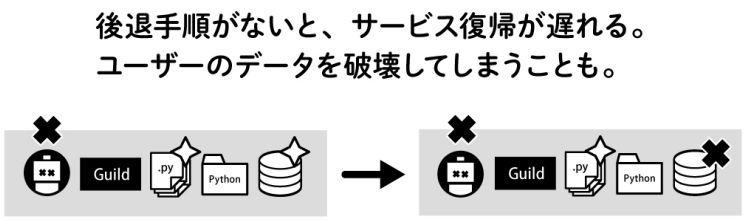
```
1．DBのバックアップをとる。
2．DBスキーマ変更：WordテーブルにGuild Snowflakeカラムを追加する
3．DBデータ操作：WordテーブルのGuild Snowflakeカラムを埋める
4．DBスキーマ変更：Guildテーブルの主キー・Wordテーブルの外部キーの制約をそれぞれ落とす
5．DBスキーマ変更：GuildテーブルのGuild Snowflakeカラムに主キー制約、
   WordテーブルのGuild Snowflakeカラムに外部キー制約をつける
6．コード変更：Guildテーブルの主キーとしてSnowflakeを参照するように変更
7．Botの起動に失敗する場合、一度DBからバックアップを戻し、データを復元する。
   6で行ったコード変更を元にもどす。
```

11.1.3.2　後退の手段を確保

アップデート手順書に書くべきこととして、「後退手順の準備」と、「後退判断のタイミングと基準」があります。リスト11.1でいうと、1が「後退手順の準備」、7が「後退判断のタイミングと基準」にあたります。

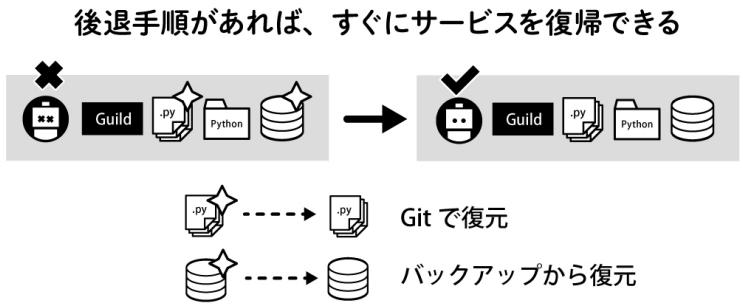
後退手順とは、行ったアップデートについて、ソースコードやDBの内容などを含め、すべてをアップデート前の状態に戻す手順です。この手順を用意していない場合、アップデートによって本番環境のサービスがサービス提供できない状態に陥った際、取り返しの付かない事態を招くおそれがあります（図11.2）。

図11.2: 後退手順がないとき



このような場合でも、後退手順があれば、DBとコードをアップデート直前の状態に戻し、すぐにサービスの運用を再開できます（図11.3）。

図11.3: 後退手順があるとき



もちろん、そもそもそのような状態に陥らないようリハーサルをするのですが、万が一の事態に備えて二重三重の準備をしておくことが、サービスの稼働率を上げることに繋がるのです。

11.1.3.3　リハーサル準備

アップデート手順書が作成できたら、なるべく本番に近い環境を用意します。本番に近い環境の例として、DBのデータがあります。本番環境のデータをすべて複製したデータをリハーサル用データとして使用します。shovelの場合、BotアカウントおよびGuildについては、開発用Botおよび開発用Guildを使います。

本番と同じデータをリハーサルに用いる理由は、データの内容、とくにデータの数が大幅に違うと、計算量や入出力待ち時間に違いが生じるためです。本番と異なる環境でしかテストを行っていなかったために、いざ本番で動かしたとき、テストでは起きなかったはずの不具合が顕在化してしまった、というケースはしばしば聞く話です。リハーサルでもこれと同じことに注意する必要があります。

11.1.3.4　リハーサルの実施

アップデート手順書通りに作業を実施し、あらためてテストを行います。この過程でアップデート手順書に誤りや記載漏れを見つけたときは、その都度修正します。

11.2　本番アップデートの実施

11.2.1　本番アップデートを実施する

importlib.reloadやdiscord.ext.commands.Bot.reload_extensionにより、再起動せずにアップデートできる場合は、リハーサルが完了次第すぐにアップデートします。

再起動が必要である場合は、ユーザーに影響の少ないタイミングにサービスの再起動を行います。サービスの内容とユーザー層にもよりますが、shovelであれば午前6～10時頃の接続数がもっとも少ないので、ここで再起動し、アップデートを行います。また、ユーザーへのリリース通知をしたい時間から逆算してアップデートの時間を決めるのもよいでしょう。もちろん、事前にDBのバックアップも必ず行いましょう。

11.2.2　動作確認をする

サービスの再起動を含め、アップデートが完了したら、まずは通常通りサービスが動いていることを確認します。無事に起動し、通常通り動作していることが確認できたら、念には念を入れ、ここでもテストを実施します。

11.3 「2系統アップデート」でダウンタイム大幅短縮！

11.3.1 大規模アップデートのジレンマ

ダウンタイムは極力短くしたいものです。しかし、サーバーのOSを変更したい場合や、ミドルウェアを載せ替えたいときなど、どうしてもサービスを長期間ダウンさせないと行えない作業というのは存在します。そんなときに検討していただきたいのが、一時的な2系統運用です。

これは、スナップショット（サーバーの内容をまるごと保存するバックアップのこと。詳細は第15章「バックアップ - 安心を確保する」で解説）を利用してまったく同じサーバーをもうひとつ立ち上げ、そこで通常の運用を継続するというものです。アップデート作業はその裏で実施します。これだけではわかりにくいでしょうから、次項をごらんください。

11.3.2 詳細な実施手順

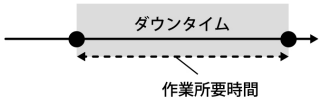
もともとあるVPSのことを、VPS①と呼びます。新しく作成するVPSのことを、VPS②と呼びます。

1. サービスの運用を停止する。
2. VPS①をシャットダウンし、VPS①のスナップショットをとる。
3. 2.で取得したスナップショットから、VPS②を生成する。
4. VPS②を起動し、サービス運用を開始する。
5. 4.で運用を開始したサービスが正常に動作していることを確認する。
6. VPS①を起動し、アップデートに必要な変更を施す。
7. VPS②のサービス運用を停止する。
8. VPS②のDBのバックアップを取得し、VPS①に転送する。
9. 8.で転送したバックアップの内容を、VPS①のDBにリストアする。
10. VPS①でサービス運用を開始する。
11. 10.で運用を開始したサービスが正常に動作していることを確認する。
12. VPS②をシャットダウンし、削除する。

11.3.3 2系統アップデートのメリット・デメリット

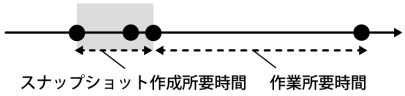
メリットはなんといってもダウンタイムの短さです。先ほどの図からもわかりますが、サービスを止めたまま作業すると、かならず作業所要時間より長いダウンタイムが発生します（図11.4）。

図11.4: 通常のダウンタイム



2系統アップデートを行うことで、大幅な時間短縮が見込めます（図11.5）。

図11.5: 2系統ダウンタイムの比較



ただし次のようなデメリットもあります。IPアドレスの一時的な変化と、サーバーを2台稼働させることによる金銭的負担です。Discord Botの場合、BotクライアントのIPアドレスが変わってもサービスには何ら影響はありません。HTTPサーバーを利用したWebアプリケーション部分など、IPアドレスにかかわる部分があれば影響を受けますが、サーバーでのリダイレクト設定やDNS設定で対応可能です。周辺部分は諦めて縮退運転するという手もあります。また金銭的負担については、VPSであれば日単位でサーバーをレンタルできるため、shovel程度の軽量なシステムであれば、1日数百円以下で済みます。

第12章　Botの土台作りのダイジェスト

　この章では、サービスが動作する土台であるサーバーについてお話しします。shovelはVPS上で動いています。なぜVPSなのか？　それはこのサービスの信頼性を低コストで確保するにはVPSがもっとも適切だったからです。そこをくわしくお話ししたあと、VPSの契約から基本のセットアップまでの過程をダイジェストで見ていきましょう。

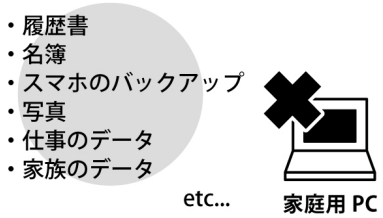
12.1　どこでBotを動かすか検討しよう

12.1.1　Bot運営にサーバーを使うわけ

Botを動かすだけなら、必ずしもサーバーが必要なわけではありません。ここでの「サーバー」とは、「サービスを提供することに主な目的をおいた、インターネットに接続されたコンピューター」のことを指します。shovelはdiscord.pyというPython用ライブラリーを使って開発されています。なので、Pythonが動いて、インターネットに接続できるなら、普通のPCでshovelを動かすことも可能です。

では、なぜわざわざサーバー代を払ってまでサーバーで動かしているのでしょうか。これには3つの理由があります。第一に、利便性の問題です。自分用のPCでBotを動かしていると、24時間PCを立ち上げっぱなしにしなければなりません。不意に停電やインターネット回線の不調が発生した場合にBotが止まってしまいますし、再起動を自由に行うこともできなくなります。第二に、セキュリティの問題です。私用PCには個人情報や重要なデータが保存されている可能性があり、思わぬ事故で情報漏洩が発生した際に被害が出ます（図12.1）。第三に、機能上の問題です。個人PCの多くはWindowsです。家庭用のWindowsOSでWebサーバーやDBサーバーを立てて安定稼働させることも可能ではあります。しかし、もともとUNIX系のソフトをWindowsに移植したものを使うことになり、前例も少ないため、インストールや設定に関する情報量の少ない傾向があります。また、個人用のWindowsのライセンスはサーバーとしての利用に制限があります。

図12.1: 個人PCの情報漏洩リスク

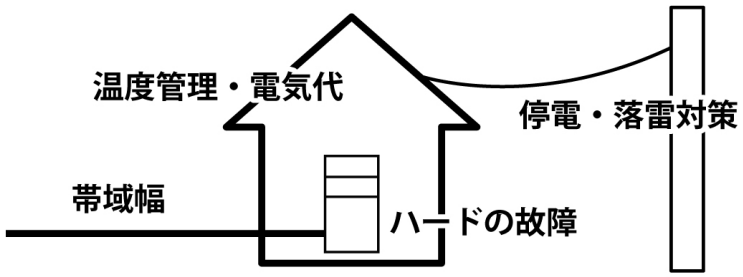


これらの理由から、本格的なBot運営をする場合は、自宅PCではなくサーバーを用意するべきだと考えます。ただし、親しい友人向けにお試しで作ってみる場合や、DBなどのミドルウェアが不要な場合などは、必ずしもこの限りではありません。

12.1.2　自宅サーバーではなくVPSを選ぶ理由

自宅サーバーは一見手軽なように見えますが、実運用に堪えるレベルのサーバーを一般住宅に設置して安定稼働させるというのは想像以上に大変です。一般家庭用のネットワークは稼働率や速度容量、固定IPアドレスが持ちづらいなどの点で十分とは言えません。また、温度管理や停電・落雷対策も楽にできるものではありません。夏場はサーバーを冷やすエアコンの電気代だけで数千円、数万円に上るでしょう。他にも、運営者の自宅が停電したときにユーザー全員がBotを使えなくなってしまったり、運営者の家族が動画を見るたびにBotの反応が遅くなったりしている可能性もあり得ます。そのような状態では、実用的なBotをユーザーに提供できているとはいえないでしょう。それに、メモリやCPUのスペックを上げると、そのたびにお金がかかります。故障への対応も自分でしなければなりません（図12.2）。

図12.2: 自宅サーバーの弱点



12.1.3　PaaSではなくVPSを選ぶ理由

自宅サーバー以外でアプリケーションを稼働させる場合、VPS以外には、HerokuやAWS等のPaaSを利用するという手段もあります。これらの挙動に精通している場合、もちろん選択肢として検討して構いません。しかし、これらのツールに触れるのもBot開発もはじめてだという方には、難しく感じてしまう可能性が高いです。なぜなら、このようなツールには普通のサーバーにはない特殊な制約が存在するからです。

何をするにもまず、ツールを掌握することが必要になります。そうでないと、うまくいかない場合に原因がどこにあるのかを切り分けにくくなります。もちろん、使ってはいけないわけではありません。うまく使えば費用を抑えつつ高いパフォーマンスを得られるでしょう。

12.2　サーバーのセットアップをダイジェストで紹介！

12.2.1　サーバーの入手 - VPSの契約

まずはサーバーを入手します。ここではVPSを契約します。

VPSを契約する際には、まずOSを選択します。契約後、選択したOSがインストールされた状態のサーバーが手に入りますので、コントロールパネルから電源を入れて起動してみましょう。また、サーバー作成前に、作成するサーバーへのSSHログインに使用する公開鍵を設定できるサービスも一般的です。これを設定しておけば、SSHでのパスワードログインを一切行わずに済むため、安全性が高まります。

12.2.2　セキュリティ設定

契約して起動が完了した瞬間から、サーバーはすでに全世界に向けて公開されています。もし初期設定を後日にまわすのであれば、セットアップが完了次第、VPSのコントロールパネルから電源を落としておきましょう。「ぽっと作っただけのサーバーなんて、誰も来ないよ」と思う方もいるかもしれません。しかし、攻撃者はIPアドレスを総当たりして攻撃を繰り返しています。

もちろん、長いパスワードを設定して破られにくくしておけば、立ち上げておくだけで乗っ取られる可能性も低くはなるでしょう。ですが、念には念を入れ、セキュリティを万全にするまで電源をオフにして攻撃不能の状態にしておくことは、決して無駄なことではありません。このセキュリティ設定は多岐にわたるため、第13章「セキュリティ - Botとユーザーを守る壁」で詳細に述べます。

12.2.3　HTTPサーバの導入

SSHでログインすれば、サーバーにアクセスできます。しかし、出先でスマートフォンしかないときにSSHを使ってサーバーの状態を確認するのは、現実的には少々厳しいのではないでしょうか。そこで普段から、もっと手軽にサーバーの状態を確認できるように、HTTPサーバーを設置しておくとよいでしょう。HTTPサーバーがあれば、サーバーから配信したコンテンツをブラウザひとつで見られるようになります。ただし、HTTPサーバーのセットアップは環境に応じて手順が異なるため、ここでの詳しい手順の解説は省略します。ご自身の環境に応じて調べてみてください。

12.2.4　HTTPS化設定

HTTPサーバーを立てたら、SSL証明書を設定しておきます。自己責任でHTTPのままでも構いませんが、ユーザーがアクセスする可能性のある場合や、パスワードを入力したりする用途があるならHTTPS化しておい

たほうが安心です。SSL証明書は、Let's Encrypt等の無料のものから有料のものまで種類も豊富なため、提供するサービスに応じて適切なものを選択しましょう。

注意点ですが、IPアドレス（xxx.xxx.xxx.xxx形式のもの）に対してSSL証明書を発行することはできません。独自ドメインは年間数百円程度の費用で簡単に発行できるので、日頃よりひとつは持っておくと、色々と融通がきいて便利です。筆者も、別件で使用しているドメインのサブドメインをVPSに割り振り、Let's EncryptのSSL証明書を適用しています。

12.2.5　監視の仕組みをととのえる

サーバーの動作が正常であることを確かめるためや、サーバースペックが過不足ないか確認するため、サーバーの監視を行います。詳細は第14章「監視 - 24時間みまもり体制」でお話しします。

12.2.6　アプリケーションのインストール

Botに必要なアプリケーションをインストールします。Linux OSへのインストールには、大きく分けて3つの方法があります。インストールしたいアプリケーションや必要なバージョンなどの情報を総合し、最適な方法を決めましょう。

1. apt、yumなどのOSコマンドでインストールする
2. バイナリファイルをダウンロードする
3. コードをダウンロードし、ビルドする

12.3　DBでデータを永続化しよう

12.3.1　DBの必要性

データを永続化する手段について考えていきましょう。ここで「データの永続化」とは、「Botアプリケーションが終了しても、Botに登録した設定など、次回以降の起動に引き継ぎたいデータが失われないようにすること」を指します（図12.3、図12.4）。shovelであれば、各Guildの辞書や読み上げ設定、ユーザーのボイス設定などが永続化すべきデータです。

図12.3: データの永続化をしていない場合

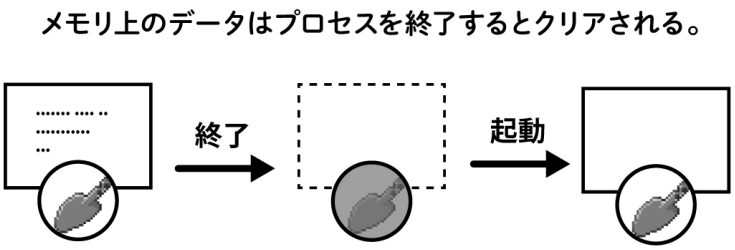
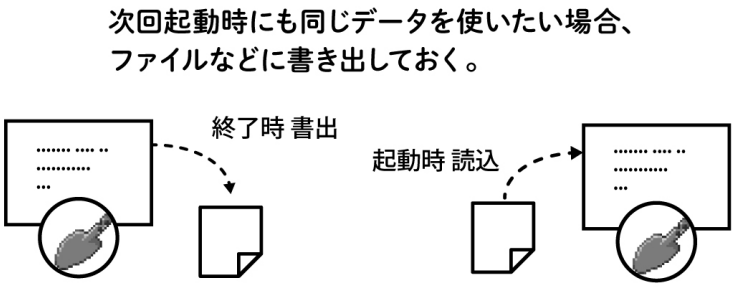


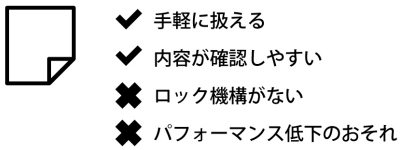
図12.4: データの永続化をしている場合



永続化する手段は主にふたつあります。ひとつはファイルへの書き込みです（図12.5）。ファイル入出力はどの言語でも簡単に行えますし、DBサーバーを用意する必要がなく、手軽に扱うことができます。また、ファイルフォーマットにもよりますが、テキスト形式のファイルであれば内容の確認もファイルを開くだけなので簡単です。しかしいっぽうで、OSレベルのファイル入出力自体にはロック機構は保証しづらいため、アプリケーシ

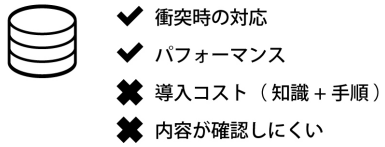
ョン内でも排他制御が必要になります。複数アプリケーションから操作することとなったとき（たとえば、Bot設定をWebインターフェイス経由で行えるようにしたくなった場合など）にも、複数アプリケーション間での排他制御を行わなければなりません。また、大量の書き込み・削除を行っているとパフォーマンスが低下する原因にもなります。

図12.5: ファイルでの永続化



もうひとつの選択肢が、DBの利用です（図12.6）。DBMS（MySQLやPostgresql等）は、「管理システム」の名のとおり、データをDBに格納するさまざまな懸念事項の処理を請け負ってくれるものです。たとえば、複数のアプリケーションから同じデータへの書き込みが発生した際にエラーを発生させたり、複数のデータの格納を行いたい際に、同一タイミングで確定させたりすることが可能です。また、大量のデータ入出力のパフォーマンスを最適化し、最低限のディスク入出力でデータの読み書きを実現してくれます。いっぽうで、利用するにはDBMSをインストールし、常時稼働させておく必要が出てくるため、単なるファイルよりも環境構築や運用のハードルが高くなります。

図12.6: DBでの永続化



まとめると、ファイル入出力は手軽に試せる反面、パフォーマンスや排他制御の点で難があります。DBは初期設定の必要があり、データの格納方法もユーザーには一見不透明ですが、入出力やデータの書き込みタイミングの衝突に対応する機構があり、信頼性とパフォーマンスが高さに期待がもてます。どちらもメリット・デメリットがあるので、用途に応じて選択しましょう。試し開発のときはまずファイルを使い、本格的な運営を始めたいときにDBへ移行するという手順を踏むというのもよいでしょう。DBの設定で行き詰まって、Bot開発自体を諦めてしまうことだけは避けたいものです。

12.3.2　DBMSの導入

PostgreSQL、MariaDBなどのDBMSを導入します。MariaDBは、MySQLから派生したオープンソースのDBMSです。

12.4　バックアップ設定

　データが不慮の事故で消失してしまわないように、バックアップをとっておきましょう。また、作成したバックアップをサーバー内に保存していてはサーバーに何かあったときに意味がないので、VPSとは違う場所に転送しておきます。詳細は、第15章「バックアップ - 安心を確保する」で紹介します。

第13章　セキュリティ - Botとユーザーを守る壁

この章では、個人開発サービスを運営する上で欠かせないセキュリティについてお話しします。実装の本筋からは外れるため、技術のチュートリアルなどで触れられることは少ない話題ですが、攻撃による被害を防ぐことはサービスの信頼性確保において絶対必要です。まずはOSのセットアップにおけるセキュリティについて述べ、続いてBotを運営する上でとくに気をつけるべき攻撃手法について解説します。

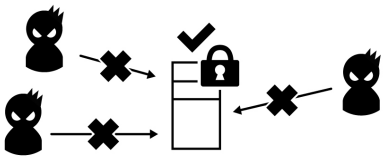
13.1　知らないうちに犯罪に加担しないために

13.1.1　「お試しだから」は通用しない

VPSは手軽にオンラインで契約が完結しますし、自宅サーバーも電源を入れて稼働を開始するだけなら、難しいことではないでしょう。しかし、サーバーを安全に運用していくのは、簡単なことではありません。サーバーを運用しはじめた瞬間から、サーバー管理人はそのサーバーを守る責任を負います。「何かあったら解約すればよい」という、安易な考えではいけないのです。

なぜなら、悪意ある第三者によるサーバーの乗っ取りを許すというのは、攻撃者の新たな攻撃を手助けすることに等しいからです。あなたのサーバーが使えなくなるだけなら作り直せばよいだけです。しかし、DDoS攻撃の踏み台にされたり、違法ファイルの倉庫にされたり、最悪の場合、あなたの法的責任が問われる事態にもなりかねません。VPSサービスによっては、そのような異常な挙動がみられるサーバーは自動的にシャットダウンしてくれる場合もありますが、それはあくまでサービスとしての最終ラインであり、本来はその前にサーバー管理者が止めるべきなのです（図13.1）。サービスを利用してくれるユーザーのデータを預かることがあるなら、なおさらです。

図13.1: サーバー管理者の負う責任



もちろん、人間なら誰しもミスはあるものです。サーバーを乗っ取られることもあるでしょう。それもひとつの経験といえるかもしれません。しかし、サーバーを守る方法についてはインターネットと書籍、いずれにも多くの知見があり、対策を打ちやすい部分でもあります。万全を期すよう努力しましょう。

13.1.2　万全はない、安心はしない

セキュリティに万全はありません。どんな技術で守っても、そのときは安心でも将来的に破られる可能性はつきまといます。万全の備えをしたつもりでいても、何かの間違いで守れてはいなかったということも、起こり得るでしょう。そのため、「大丈夫なはず」がひとつふたつ崩れても問題ないようにしておくことが重要です。例を以下に示します。

・機密データは、自分以外にアクセスできないはずの場所に置いたうえで、万全を期して暗号化しておく

・DBにおいて、特定のテーブルにしかアクセスせず、読み込みのみ行うアプリケーションには、該当テーブルの読み込み権限のみを持たせる
・絶対に漏れてはいけないデータはそもそも保持しない

13.1.3　セキュリティリスクの管理

前項と矛盾するようですが、サービス運営するうえでは、絶対に漏れてはいけないデータを保持せざるを得ない場面もあります。そこで大切になってくるのがセキュリティリスク（以下、この項ではセキュリティリスクのことをリスクと呼びます）の管理です。

まず、アプリケーションと、アプリケーションにかかわる要素（ユーザー、DB、バックアップサーバー、etc）をリストアップします。そして、それらのインターフェイスを順番に整理します。その各インターフェイスに関して、どのようなリスクがあるかを洗い出します。洗い出したそれぞれのリスクについて、発生する見込み、発生した際の被害の大きさから評価し、対応を決めます。リスクは、ある程度受容することも大切です。杞憂といえるような心配事についてまですべて予防策を講じようとすると、多大なコストがかかります。そのようなケースについて、予防はせず発生時の対策のみ決めておくこともリスク管理のうちです。本筋のサービス開発にリソースを確保することを忘れないようにしましょう。

13.2　サーバーのセキュリティを固めよう

VPSにしろ、自宅サーバーにしろ、これだけやれば絶対安心ということはありません。ですが、サーバーのセキュリティを高めるために行うべきことは共通しています。ここでは、最低限行うべきセキュリティ対策の手順を紹介します。

13.2.1　パッケージのアップデート

OSをセットアップしたら、付属ソフトウェアのバージョンが古いままになっていないか確認しましょう。バージョンが古いと、既知の脆弱性を突いた攻撃に遭う危険性も高まります。パッケージ管理ツール（apt、yum等）を使い、アップデートするようにしましょう。パッケージ管理ツールを使っても万全であるとはいえません。とはいえ、ひとつひとつ確認するのは大変なので、［使用しているOSとバージョン］＋［脆弱性］等で検索するなどして、対応すべき脆弱性がないかを調べるとよいでしょう。

13.2.2　プロセス設定

OSコマンドのpsを使い、デフォルトで動いているプロセスを確認します。そして、OSの起動スクリプトを確認し、自分のサービスを運営する上で不要なプロセスは動かないようにしておきましょう。このとき、psコマンドで出てくるプロセスの顔ぶれをよく見て、見慣れておくとよいでしょう。もちろん、すべてのプロセスの挙動を完全に把握するに越したことはありませんが、それはすぐには難しいことです。せめてサーバーに不審な挙動がみられたとき、プロセス一覧に見慣れないものが紛れていることを見抜ける程度には、通常動いているプロセスを見慣れておくことを推奨します。

また、いわゆるウェルノウンポートで動作しているプロセスや、デフォルトのポート番号が広く知られているプロセスについて、ポート番号設定をデフォルト以外へ変更しておくことを強くおすすめします。たとえば、「mysql」「sshd」などがそれにあたります。どれだけ面倒でも、「sshd」だけはかならず変えておきましょう。sshdログインを使った攻撃の数が激減します。

13.2.3　ユーザーの作成

ルート権限を必要としないような作業は、別途作成したユーザーで行うのがよいでしょう。また、セキュリティのレベルにもよりますが、可能であれば作業用ユーザーがOSコマンドのsudoを使えるようにしておくと便利です。sudoコマンドをインストールし、OSコマンドのgpasswdを使って作業用ユーザーをsudoグループに追加しておきます。

13.2.4　rootログインの禁止

13.2.4.1　rootログインを禁止する意義

UNIX系のサーバーでは、rootログインは不可に設定しておくのが一般的です。その上で、作業をするためには、まず通常ユーザーでログインし、root権限の必要な操作のみsudoを利用して実行します。その理由はふたつあります。

まず、rootログインの可能な状態だと、攻撃者にサーバーを乗っ取られやすくなります。ユーザー名が分かっていてシステムの全権を握れるrootは、攻撃者にとって絶好の標的であるため、真っ先に破ろうとしてきます。rootログインを不可に設定しておくと、このポイントを破られる可能性をゼロにできます。

次に、root権限についてですが、これはシステムの破壊すら可能な絶大なものです。操作ミスは絶対に許されません。作業するにあたり、ログインするたびにそんな権限を持つ必要がそもそもないのです。この場合は、通常ユーザーでログインして、root権限を必要とするOSコマンドのみ明示的にsudoを利用し、root権限を付与して実行するのがよいでしょう。

rootログインを禁止するには、次のような設定が有効です。

・rootのシェルを変更しておく
・sshのrootログインを禁止しておく
・TTYのrootログインを禁止しておく
・PAMのrootアクセスを禁止しておく

13.2.4.2　sudoが使えるならrootをとられるのと同じでは？

ここまで読まれた方は、「いくらroot権限を取られないようにしたとしても、OSコマンドのsudoを使えるユーザーとしてログインされてしまったら、root権限を取られるのと同じくらいまずいのではないか？」と思われるかもしれません。たしかにその通りです。ではなぜ、ことさらrootログインを防ぐのでしょうか。ここは、攻撃者の気持ちになって考えてみましょう。

まずあなたは、サーバーAにSSHログインを試みようと、sshdが待ち受けるポート22へ向かいます。しかしこのポートは封鎖されていました。なんとか総当たりでsshdのポートを探し当てたあなたは、rootでのパスワードログインを試行します。しかし、サーバーAはrootログインを禁じており、まだアカウント名も未知であるほかのユーザーでしかログインできないようです。そのうえ、パスワードログインは受け付けてくれず、数百文字以上の文字列からなる暗号鍵を合致させないと入れないことがわかりました。なんとかユーザーとしてログインできても、サーバーAのroot権限を取れるとは限らないと思ったあなたは、サーバーAを諦め、サーバーBに向かいました。するとどうでしょう、なんとこのサーバーB、rootでのSSHパスワードログインを許可しています。あなたは辞書型攻撃でパスワードを破り、無事サーバーBのroot権限を取ることができました。サーバーAとサーバーB、どちらがroot権限を取られやすいか、言うまでもありませんね。

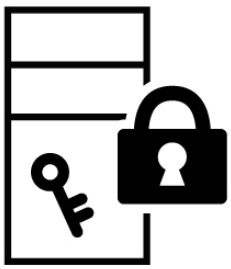
ご理解いただけたでしょうか。もちろん、実際の攻撃は、人の手ではなく自動化されたプログラムによって行われているため、この例のように「根負け」ではなく「破るための必要な計算量が多すぎる」という形で諦めることになります。たとえるなら、人感センサーのライトに照らされても人間は平気ですが、人感センサーのついている家をあえて狙う泥棒が少ないのと同じです。これがrootログインを禁じる理由です。

13.2.4.3　キー閉じ込めに注意！

SSHのrootログインを禁止する際には注意が必要です。まずは、作業用ユーザーのSSH設定を行い、実際に作業用ユーザーとしてSSHログインできることを確かめます。そしてその後、rootユーザーによるログインを禁止するよう、設定を変更しましょう。

これを怠ると、作業用ユーザーとrootユーザーのどちらでもログインすることができなくなり、図13.2のように、まるで車の中にキーを閉じ込めてしまうような事態になってしまうのです。ちなみに筆者も一度これをやってしまい、設定の初期段階だったので、諦めてVPSを作り直したことがあります。筆者のようにVPSを作り直さなくても、VPSサービスのダッシュボードからコンソールにログインする方法などでなんとかリカバリーできることもあります。ですが、はっきり言ってとても面倒な作業です。

図13.2: サーバーのキー閉じ込め？



13.2.5　SSHの設定

13.2.5.1　ポート番号変更

「プロセス設定」の項でも述べましたが、sshdだけはかならず使用するポート番号を変更しましょう。SSHのポートを22のままにしておくと、一晩で数千回のログイン試行を受けることになります。ちなみに、これは筆者がはじめて立てたVPSで実際に経験したことです。

13.2.5.2　パスワードログインは厳禁

OSによっては、sshdの設定でパスワードログインがデフォルトで「yes」になっていることもあるでしょう。これをそのままにしておくことは厳禁です。OSのログインパスワードは、長くてもせいぜい数十文字でしょう。それに対して、暗号鍵ファイルの長さは、暗号化方式にもよりますが、数百文字以上です。もちろんセキュリティの堅牢さは鍵の長さだけに依存するものではありませんが、総当り攻撃を想定した場合、数十文字のほうが圧倒的に破りやすいことは誰の目にも明らかです（図13.3）。

図13.3: パスワードと暗号鍵ファイルの比較

13.2.6　ファイアウォールの設定

各種サービスのポート設定ができたら、パケットフィルタリングを設定し、不要なポートへのアクセスは完全に遮断してしまいましょう。基本的にすべてのパケットを破棄するようポリシーを設定し、各サービスで使うと設定したポートのみを開放しておきます。

注意すべき事柄として、ここでもSSHログインの鍵閉じ込めは起き得ます。sshdのポートを22から新ポートに変更したのち、ファイアウォールでポート22を閉じ、新ポートを開放して適用したつもりで再起動をします。しかし、設定ミスによりsshdのポートが22のままになっていると、SSHログインができなくなってしまいます。実は、なんと筆者はこれもやってしまったことがあります。

13.2.7　Logwatchの導入

事前にセキュリティを高める設定ではありませんが、何かあったときにすぐ気がつけるよう、logwatchを導入し、日次メールレポートを送付するよう設定しておくとよいでしょう。logwatchは、サーバー上で動作するさまざまなサービス（cron、SSH、HTTPなど）のログを横断して解析し、人間が読みやすい形式のレポートにまとめてくれるアプリケーションです。「昨日」「過去1か月」「X月X日～X月X日」といった形で時間帯を指定すると、その範囲のレポートが取得できます。

13.3　Botを狙う攻撃手法を知ろう

13.3.1　Botに特徴的な脆弱性

Discord Botは、サーバー内部を直接ユーザーにさらけ出しているわけではありません。しかし、ユーザーからの入力を扱うという性質上、「ReDoS脆弱性攻撃」「SQLインジェクション」「OSコマンドインジェクション」「ディレクトリトラバーサル」の4つの攻撃には注意が必要です。

13.3.2　ReDoS脆弱性攻撃

ReDoSは、特殊なパターンで照合処理を行うことにより、照合対象文字列が長くなるにつれ、処理時間が爆発的に増加することを利用した攻撃です。

Botの機能によっては、ユーザーが入力した正規表現パターンを扱うこともあるかもしれません。そんなときは「re2」を使いましょう。re2は、Googleによって開発された、ReDoS脆弱性攻撃への対策を行った正規表現モジュールです。さまざまな言語でのラッパーが公開されており、Pythonにもラッパーモジュール「pyre2」があります。

re2では、入力文字列の大きさに対して処理時間が線形となるよう保証されています。re2が目指すのは爆発的に処理時間が延びないようにすることですので、同じ入力に対しPython標準モジュール「re」のほうが高速となることもあります。ReDoSのおそれがない場合など、状況に応じて使い分けましょう。

13.3.3　SQLインジェクション

ユーザーが入力した文字列をSQLクエリに組み込むつくりのアプリケーションで、ユーザーが入力した文字列によっては、意図しないSQLクエリが実行されてしまう脆弱性をついた攻撃です。

PythonでSQLを利用するモジュールを使っていれば、モジュール側で引数部分を自動的にエスケープしてくれることが多いです。しかし、思わぬ事故を防ぐためにも、自分の利用しているモジュールがどのようなSQLインジェクション対策をしているかは把握しておく必要があります。

13.3.4　OSコマンドインジェクション

引数などとしてユーザーが入力した文字列をシェルコマンド文字列に組み込むつくりのアプリケーションで、ユーザーが入力した文字列によっては、意図しないシェルコマンドが実行されてしまう脆弱性をついた攻撃です。

ユーザーの入力した文字列を使ってコマンドラインを操作するときは、ユーザーが入力した文字列を適切にエスケープする必要があります。とはいえ、攻撃の手法は多岐にわたるため、手動で実装するのは好ましくあ

りません。Python標準モジュール「subprocess」は、OSコマンドインジェクションへの対策を意識して設計されています。ドキュメントを読むと、runおよびPopen関数のパラメーターにshellがあり、これがFalseであれば自動的にエスケープが行われる旨の解説があります。また、shell=Trueの場合であっても、shlex.quote()関数を使えば適切にエスケープが行える旨も解説されています。

13.3.5　ディレクトリトラバーサル

ユーザーが入力した文字列をファイルパスとして展開する処理の脆弱性を突き、本来アクセスできないはずのファイルにアクセスする攻撃です。ユーザーが指定した文字列をファイル名にするのではなく、ユーザーのIDや、メッセージのIDなどでファイル名を作成するようにすることが対策となります。また、アプリケーションの実行ユーザーに、データを配置するフォルダーより上位のフォルダーへのアクセス権限を持たせないことも対策のひとつです。

FlaskやDjangoなどのWebアプリケーションフレームワークには、ディレクトリトラバーサルを引き起こさないパス結合関数safe_joinが用意されています。

13.3.6　その他気をつけるべきこと

本節では、Botがとくに注意すべき攻撃手法について紹介しました。Webアプリケーションへの攻撃手法について学ぶ際には、徳丸浩『安全なWebアプリケーションの作り方　脆弱性が生まれる原理と対策の実践 第2版』（2018　SBクリエイティブ）がおすすめです。

またBotを攻撃から守るには、Bot自身のコードだけでなく、Botから利用しているOSコマンドや外部パッケージの脆弱性についても日頃から気にかけておく必要があります。

shovelを例として説明します。shovelでは、読み上げを行うためにユーザーが入力したメッセージのテキストを音声合成エンジンに標準入力経由で渡す処理があります。万が一音声合成エンジンに脆弱性が発見された場合、その処理が攻撃の起点にされるおそれがあり、すみやかに対応する必要があります。音声合成エンジンに限らずshovelから利用しているOSコマンドなどの脆弱性情報には、日々気を配っています。

第14章　監視 - 24時間みまもり体制

　サービスを動かしていると、「ちゃんと動いているんだろうか？」「サーバースペックは過不足ないだろうか？」といった心配が出てきます。この章では、サービスの運用状態を監視し、その結果を蓄積し、運営に活かす方法についてお話しします。個人開発サービスでも手軽に扱える無料の監視ツールについても紹介します。

14.1　みまもり＋非常アラート＝監視

14.1.1　監視の概念

Botがうまく動いているか心配なとき、あなたは何をするでしょうか？　エラーが出たらアプリケーションログを出力するよう実装してターミナルをじっと見つめていれば、すぐにエラーの発生に気付けますね。しかし、あなたが24時間そうやってターミナルを見つめ続けることは不可能です。

そこで、あなたはBotのエラー発生状況を計測・記録するプログラムを作るかもしれません。そして、エラーが起きたら大きな音でブザーが鳴るしくみを作るかもしれません。このように「Botの動作状況を見守り、障害発生を素早く検知し対応する体制でダウンタイムの最小化を図る」ことが監視の目的です。

本書では、監視という言葉について、以下のように定義します。

・規定された項目について、データを収集する。必要であれば、収集したデータを時系列順に蓄積する。

・データを利用して、条件に基づいてアラート（警告・ブザーなど目立つ通知をすること）をあげる。

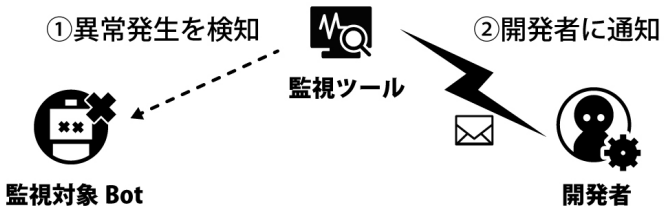
14.1.2　監視の目的

なぜBotを監視する必要があるのでしょうか。理由はいくつかあります。

14.1.2.1　異常をみつけるため

致命的なエラーが発生し、Botが利用できなくなったとしたら、あなたならどうしたいですか？　状況が許すなら、すぐに対応して正常な状態にしたいと思うのではないでしょうか。適切な監視体制を作っていれば、これが可能になります。「Botが正常な状態ではない」ことを判定するしくみを作り、この判定がONになったら何らかの方法でアラートを上げるようにしておきます（図14.1）。アラートの方法としては「メールを送る」「ブザーを鳴らす」などが挙げられます。

図14.1: 異常をアラート通知する



14.1.2.2　目と心を離すため

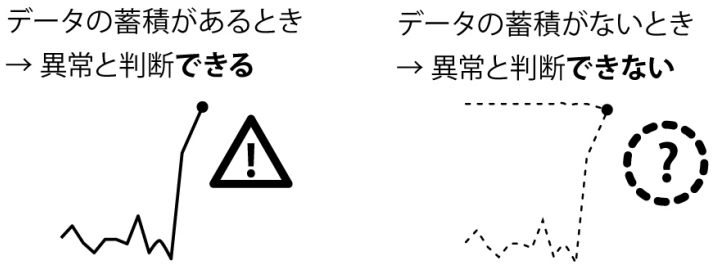
矛盾するようですが、監視のしくみを整える目的は、Botから目を離すためです。適切な監視体制が整っていれば、自分がターミナルを見つめ続ける必要はなくなります。「エラー発生時にはアラートが上がって、通知が来る」とわかっていれば、Botの様子を5分おきにチェックする必要もなくなるでしょう。「なにか起きているのでないか」とそわそわしたり、不安にとらわれたりすることもなく、本来すべきことに集中できるのです。開発の時間と心の余裕を確保するためにも、監視のしくみを整えるとよいでしょう。

14.1.2.3　データを利用するため

ユーザーに「最近、Botの調子が悪いように思う」と指摘を受けたら、あなたはどうしますか？　自分もそのBotを毎日ヘビーに使っているとしたら、自分の使い心地から、「たしかにそうかも」だったり「気のせいだよ」だったり、何かしら判断できるかもしれません。しかしそうでなかったとしたら、ユーザーの報告のとおり本当に調子が悪いのか、それともユーザーの気のせいなのか、判断がつきません。さらにいえば、自分が問題なく使えているとしても、他の環境でもうまく動いているとは言い切れません。

このようなときのために、Botの状態を表すデータを定常的に監視し、データとして蓄積しておきます。たとえば、あなたに毎日体温を測る習慣があれば、発熱時にいつもと違うことが定量的にわかります（図14.2）。また、毎日測っていれば、微熱にもすぐに気付けるでしょう。システムについても同じことがいえます。

図14.2: 普段からデータをとっておく重要性

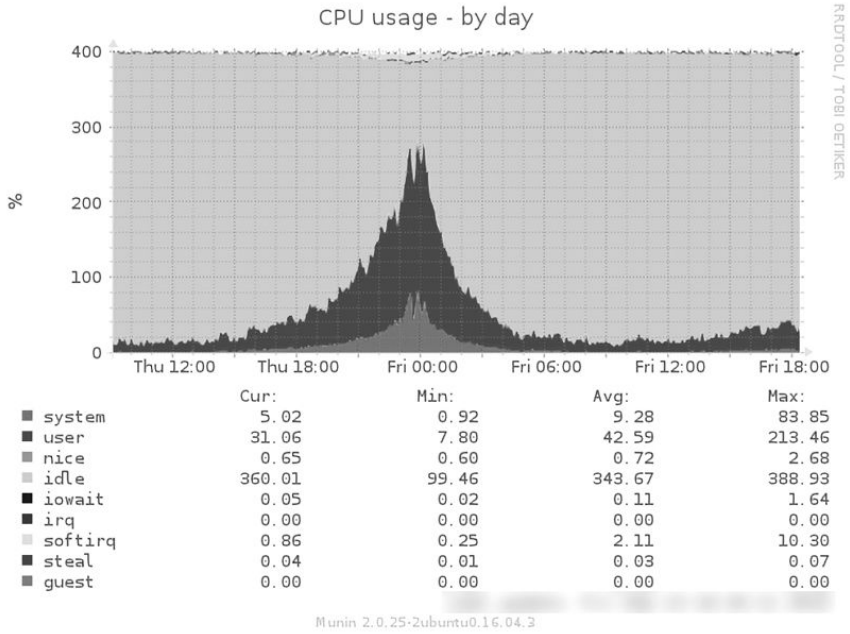


14.2　何を監視する？

14.2.1　システムの何を監視するか

一般的なサーバー監視では、ネットワーク、DB、メモリやCPUなど、インフラやミドルウェアの状態を監視します。これらを監視しておくことで、リソースが逼迫していることや、逆にスペックが過剰であることをすみやかに発見できます。また、Botに障害が発生した際には、アプリケーションに問題が発生したのか、OSで問題が発生したのかを切り分ける必要が生じます。このとき、システムの監視が役に立ちます。図14.3は、図14.4と同じ時間帯のCPU負荷グラフです。CPU負荷は、読み上げメッセージ数と文字数と同期していることがわかります。

図14.3: CPU使用率の監視グラフ

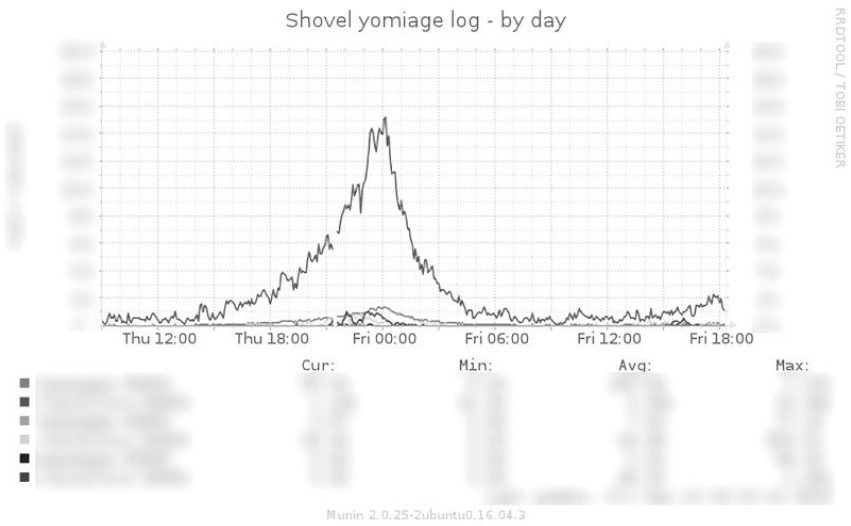


14.2.2　サービスの何を監視するか

システム監視の他に、サービス特有の数値を監視することも必要です。Discord Botであれば、基本的には、参加Guild数、延べユーザー数、ユニークユーザー数を監視対象とするとよいでしょう。これらの指標から、Botがどれくらい使われているかがわかります。また、発生したERRORログ、WARNINGログの数を監視するのもよいでしょう。Botの調子が悪いとき、それらが通常に比べてどのような数になっているかを見ることで、定量的に把握できます。

shovelでは、これらに加え、単語登録の総数、声設定の総数、直近の読み上げメッセージ数と文字数、ボイス接続数を監視しています。単語登録の総数を監視することで、shovelがどれくらい活発に使われているかがわかります。声設定の総数を監視すると、shovelを楽しんでくれているユーザーの数がわかるでしょう。読み上げメッセージ数と文字数・ボイス接続数からは、shovelにかかっている負荷がわかります。図14.4は、「読み上げメッセージ数と文字数」のMuninグラフです。0時付近に負荷のピークがあるということや、午前中はほとんど読み上げが行われていないことが一目でわかりますね。

図14.4: 読み上げメッセージ数と文字数の監視グラフ



14.3　監視に役立つツール

14.3.1　サーバー監視はツールに頼ろう

ここでは監視のためのツールを紹介します。サーバーを運用するなら、サーバー監視は必ず実施することです。皆がやることですから、既存のツールも充実しています。それらのツールに助けてもらうのが楽に、かつ効率良く監視を行うコツといえるでしょう。いずれのツールも、CPUやメモリなどの基本的なシステムリソースに関連する状態や数値だけでなく、そのサービス独自の数値（たとえばshovelなら利用ユーザー数や接続Guild数など）を監視する仕組みを備えています。

14.3.2　Munin

Muninは、データロギングツールRRDtoolのフロントエンドアプリケーションです。監視対象サーバーにクライアントサービスをインストールするだけで監視をはじめることができます。監視によって収集したRRDtoolデータは、Webサーバーを経由し、どこからでも見やすく閲覧できます。また、プラグインの仕様も単純明快で、ユーザーが監視したい項目を自由に増やせます。監視項目の値が規定の閾値を超えたらアラートを上げることもでき、メールの送信で通知できます。カスタマイズすればWebhookでの通知も可能です。

Muninサービスの配置についてですが、監視対象とは別のサーバーを用意するか、または、監視対象サーバーにMuninのホストサービスを配置することになります。監視対象とMuninサービスを同一サーバーに同居させる構成だと、監視対象サーバー自体に不具合が発生した場合、なんのアラートも発生せず、すべてのデータがとれなくなってしまうのが弱点です。

14.3.3　Mackerel

Mackerelは、株式会社はてなが開発したサーバー監視SaaSです。Muninと同様に、監視対象サーバーにクライアントサービスをインストールするだけで監視をはじめることができます。監視データは洗練されたデザインのWebサイトで閲覧できるほか、アラートの設定などのカスタマイズもWeb上で実施可能です。ユーザーの任意の項目を監視するチェックプラグインを作成することもできます。Muninと比較しての利点として、Muninのデータ更新が5分おきであることに対し、Mackerelは1分に1回データ更新を行います。

Mackerelには無料プランが存在し、10項目まで監視できます。しかし、10項目ではサービス運用には足りません。Muninからの移行サポートも充実しているので、まずはMuninを利用し、サービスの規模が大きくなりより本格的な監視を行いたくなったタイミングでMackerelへの以降を検討するとよいでしょう。

14.4　Botをツールから監視できるようにしよう

14.4.1　Botのデータを監視するには

ここでは前節で紹介した「Munin」などの監視ツールに、サービス独自の数値を監視させる方法についてお話しします。たとえば、Botが導入されているGuildの数（以下、Guild数）をMuninで監視するにはどうしたらよいでしょうか。Botプログラム内では、Guild数はリスト14.1のように短いコードで簡単に求められます。

リスト14.1: 準備完了時、Guild数をCLIに表示するサンプルコード

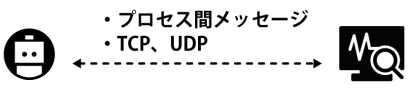
```
import discord

token = "XXXXX"

client = discord.Client()


@client.event

async def on_ready():

    print(f"current guild count: {len(client.guilds)}")


client.run(token)
```

つづいて、この数値をどのように監視プログラムに受け渡すかを考えていきます。これには、いくつかの方法があります。

14.4.2　プロセス間通信で監視

プロセス間通信による受け渡しは、もっとも外部要因の影響を受けにくい監視方法です（図14.5）。メリットは、アプリケーションと監視プログラムの間の通信が生きていれば必ずデータをとれることと、外部データソースを経由する方法と比較して、よりタイムラグのないデータがとれるという点です。

図14.5: プロセス間通信で監視



プロセス間通信の方法はいくつかの選択肢があります。監視は重要なシステム要素であるとはいえ、あくまでサービスの運用をより便利にするための補助的な機能です。選択肢のなかから、できるだけコストのかからない方法を選びましょう。プロセス間メッセージによる通信は、OSの仕様に依存する部分が大きく移植性も低いため、コストのかかる可能性があります。あまりコストをかけず、簡易的に通信できる手段としては、ソケット通信、とくにUDP通信が手軽に扱えるでしょう。

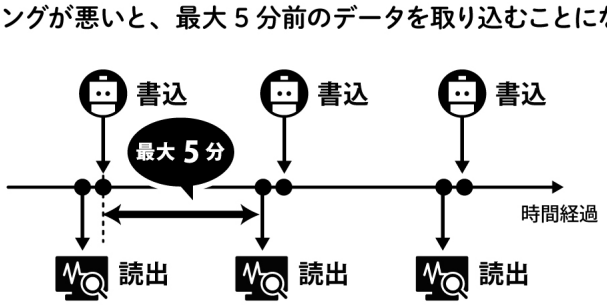
14.4.3　外部データソース経由で監視

DBや共有メモリ、ファイルなど、アプリケーションの外部にデータを書き込み、監視側と直接のつながりを持たないようにする方法もあります（図14.6）。テキストファイルやバイナリファイル、共有メモリは手軽ですが、ロック機構を導入する必要があります。可能であれば、DBを利用するのが手軽でしょう。

図14.6: 外部データソース経由で監視

この方法で注意すべき点は2点あります。1点目は、DBMSがダウンするなど、依存する外部データソースに不具合が発生した場合、データの蓄積が停止してしまうことです。2点目は、タイミングによっては監視データにタイムラグが発生することです。たとえば、5分おきに監視対象プログラムからデータが書き込まれ、5分おきに監視プログラムがデータを読み込むとします。このとき、もっとも悪い場合、最大5分近いタイムラグが発生してしまいます（図14.7）。これは、リアルタイムにデータを反映したい監視対象には不向きです。

図14.7: 監視データにタイムラグが発生する理由



shovelでは、読み上げ文字数や導入Guild数の監視について、この方法を採用しています。分単位で実態に追従する必要がないためです。

第15章　バックアップ - 安心を確保する

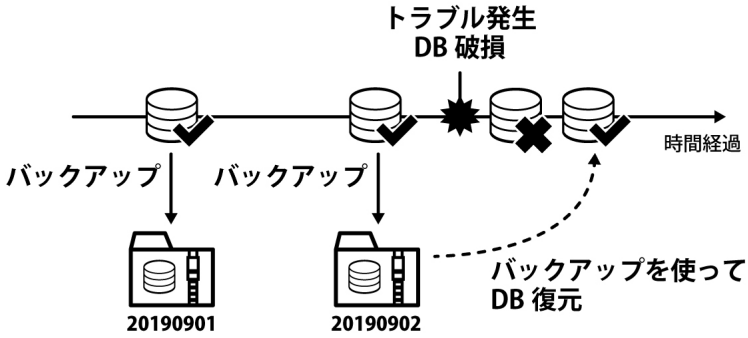
　個人開発サービス運営にはトラブルがつきものです。トラブルは起きないのが一番ですが、備えがあれば安心です。この章では、サービスを運営するうえで欠かせない、各種データのバックアップについて述べます。

15.1　バックアップの必要性

15.1.1　バックアップとは

　バックアップとは、ある時点での対象の状態を保存しておき、何かあったとき復元できるようにすることをいいます。たとえばDBのバックアップを定期的にとっておくと、削除してはいけないテーブルを誤って削除してしまった際などに、バックアップしておいたDBからテーブルを復元できます（図15.1）。

図15.1: バックアップ



15.1.2　バックアップがなかったら？

　ここで、もし定期的にバックアップを取っていなかったらどうなるかを考えてみましょう。データが損なわれたことに気づいたあなたは、「いつか取ったスナップショットやDBバックアップがあるはずだ、あれはどこに置いたっけ」と懸命に思い出しながら、大あわてで探してきて復旧することになるでしょう。復旧までに何時間もかかるかもしれませんし、大事なユーザーのデータはずいぶん前の状態に戻ってしまい、二度と取り戻すことはできません。バックアップが見つかればいいですが、最悪、サービスの継続が不可能になります。

15.1.3　バックアップをとっておこう

　このような事態を防ぐために、計画的なバックアップを行います。万が一データが損なわれたときにも、復旧までの時間を最小化することが目的です。定期的なバックアップの取得と素早いリストア手順の確立は、信頼性確保のため、継続的に運用するシステムには必須の仕組みといえるでしょう。

15.2　さまざまなバックアップ

バックアップにはさまざまな粒度・種類があります。どれかひとつをやっていれば安心というわけではなく、それぞれのバックアップの用途を理解し、システムに必要なバックアップをもれなく行うようにしましょう。

15.2.1　コードのバックアップ

これはサービス運営をしている開発者の大半が行っているものと思われます。Gitでバージョン管理し、GitHubにリポジトリを作っていればバックアップできているといえるでしょう。ありがちなミスが、「開発版だからローカルのリポジトリにしかコミットしていなかった」というものです。開発版だからリモートリポジトリに送らないでいると、マシンになにかあったとき、復元できません。開発用ブランチを切るなどしてリモートリポジトリへ送るようにしましょう。もちろんこのリモートリポジトリは、ローカルリポジトリとは別の場所に作成したものである必要があります。

15.2.2　サーバーの各種設定のバックアップ

サーバーにインストールしているミドルウェアやサービス、たとえばnginxやMySQLなどの設定もバックアップしておきましょう。サーバーに接続できなくなったときや、誤って設定を変更してしまったときに、復元できるようにしておくことが大切です。設定が完了次第そこから変更しないという場合は、設定後サーバー外部に設定ファイルをバックアップしておきます。いっぽうで、性能調整などのために頻繁に更新するのであれば、cronなどを使い、定期的にバックアップをとるようにしましょう。

15.2.3　DBのバックアップ

サービスを運営する上でもっとも注意して扱う必要があるのは、ユーザーのデータです。これは、「漏洩」「消失」このどちらも絶対に防がないといけません。自分の管理するDBにあっても、そのデータは自分のものではありません。ユーザーの大切なデータを預かっているという意識を持ちましょう。漏洩についての対策は第13章「セキュリティ - Botとユーザーを守る壁」で触れていますので、ここでは「消失」をどのように防ぐかを考えます。

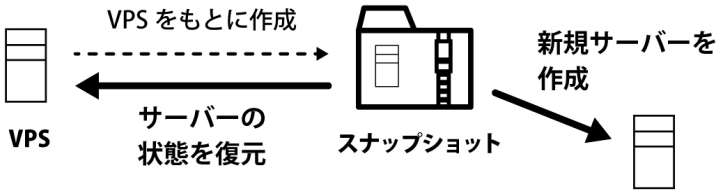
DBに格納したデータは、サーバーのデータ消失、プログラム上の誤り、メンテナンスの操作ミスなど、さまざまな不慮の事故による消失のリスクに常に晒されています。この対策として、バックアップをとることが有効です。しかし、毎日すべてのデータをバックアップしていては容量を使いすぎてしまいます。日次バックアップでユーザーがよく更新するデータをバックアップし、週次バックアップですべてのデータをバックアップするなど、頻度と対象を検討して設定しましょう。たとえば、RPGゲームのような機能を提供するBotのように、ユーザーデータ

が大幅にロールバックすると大きな損害が出る場合であれば1時間、またはもっと細かい周期でバックアップを作成してもよいかもしれません。

15.2.4　スナップショット

　これはその他のバックアップとはすこし毛色のちがうものです。仮想サーバーの場合、サーバー全体のバックアップとして使用できる「スナップショット」を取得できます（図15.2）。

図15.2: スナップショットとは



　これは、VPSやレンタルサーバーであれば管理用メニューからボタンひとつで取得できることが多いです。何かあった際にサーバーをまるごと復元できるという、最後の砦としての安心感を得られます。

　スナップショットの取得にはシステムの停止が必要であることもあり、大半のレンタルサービスでは、スナップショットの保管にはデータ容量に対応した月額料金がかかります。スナップショットの作成頻度や保存期間は、システムの運用形態と運営資金に応じて判断しましょう。

15.3　バックアップの作法

15.3.1　バックアップのスケジューリング

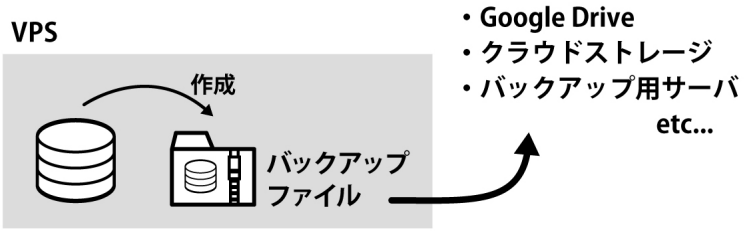
　バックアップは定期的に決まった内容を行うものであり、うっかり忘れることがあってはならないものです。このような作業を確実に行うためにはサーバーに肩代わりさせるのが一番なので、cronを使います。バックアップ用のシェルスクリプトを作成し、OSコマンドのcrontab -eで登録します。

　スナップショットのようにサーバー管理者が手作業で行うしかない作業は、予定アプリで管理するなど、忘れない仕組みを作りましょう。

15.3.2　バックアップしたデータの保存先

　作成したバックアップをサーバー内に保存していては、サーバーのデータにアクセスできなくなると取り出せなくなってしまい、バックアップとしての意味を成しません。バックアップファイルは、図15.3のように、サーバーの外部に保存しておく必要があります。shovelでは、バックアップの転送先をGoogle Driveにしています。Google Driveへのアップロードには、Google Drive用CLIツール「gdrive」が便利です。

図15.3: バックアップはサーバーの外部で保存する



第16章　運営 - ユーザーと接しよう

　個人開発サービスの醍醐味は、ユーザーと直接触れ合えることではないでしょうか。この章では、技術以外の面からサービス運営についてお話しします。shovelの運営方針についても紹介しますが、あくまで参考程度に読んでいただき、この章の内容に縛られることなく、あなたのサービスに合った運営方法を見つけてください。

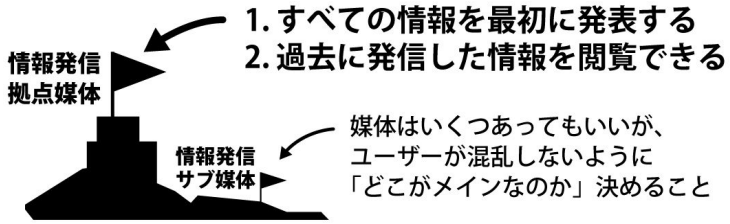
16.1　情報発信の拠点をつくろう

16.1.1　情報発信の拠点

サービスを運営するうえで、ユーザーに情報を展開するための情報発信の場を持つ必要があります（図16.1）。Webサービスであれば Webサイト自体が情報発信の場となりますし、WebサイトとTwitterアカウントなど、複数の情報発信の場を設けてもかまいません。このとき、情報発信の拠点をどこにするかをはっきり決めておきましょう。情報を受け取るユーザーが、「情報が複数の媒体に散らばって発信されており、どこを見たらいいのかわからない」という状態になることは避けましょう。

情報発信の拠点では、すべての重要な情報をまっさきに発信するようにし、過去の情報も履歴として見られるようにしておきます。拠点でない情報発信媒体では、拠点で発信した情報をピックアップして発信します。もちろんすべての情報を発信してもかまいません。また、媒体限定のキャンペーンなどについてはこの限りではありません。

図16.1: 情報発信の拠点



16.1.2　公式Guildの概要

shovelの情報発信の拠点は、Discord上のshovel公式Guildです。公式Guildのチャンネルの構成を、リスト16.1に示します。

リスト16.1: shovel公式Guild チャンネル一覧

```
shovel公式Guild
├── NEWS
|    ├── #はじめに
|    ├── #アップデート
|    └── #おしらせ
```

```
└── COMMUNITY
    ├── #質問
    └── #困ったこと
```

NEWSカテゴリ以下には開発者だけが書き込めるチャンネルがあり、shovelについての情報発信に使います。COMMUNITYカテゴリ以下にはGuild参加者全員が書き込めるチャンネルで、ユーザーからshovelについての質問や要望を挙げていただくためのチャンネルです。

16.1.3　不要な通知は極力ユーザーに送らない

情報発信を目的としたGuildであるという性質上、大切なときだけ通知を送るようにしています。Guild参加時のウェルカムメッセージはオフにしているほか、デフォルトではメンション以外の投稿はユーザーに通知しないよう設定しています。

また、Discordでは、「@everyone」という文字列を含めた投稿をすることでGuild参加者全員にメンション通知を送れます。メンション通知を受け取ると、Guildのアイコンに図16.2のように印が付きます。

図16.2: メンションを受け取ったときの通知表示

shovel公式Guildでは、この「@everyone」の使用タイミングを以下のように限定しています。

・shovelが大きなアップデートを行ったとき
・コマンド変更など、shovelの使い方に変更が生じたとき
・shovelが使用できないような、重大な不具合が発生したとき

些末なことで通知を連発していると、大事なお知らせも見逃されてしまいます。@everyoneメンションの用途を適切に限定することで、「shovel公式Guildに通知マークが出ているときは、大事なお知らせがある」とユーザーに感じてもらえるでしょう。

16.1.4　雑談チャンネルは用意しない

意見が分かれるところではありますが、shovel公式Guildはあくまで情報発信・サポートを目的としたGuildとして運営しています。

shovel公式Guildでは、shovelに関係しない投稿は一切ご遠慮いただいています。Guild運営開始直後、雑談チャンネルを用意してほしいという要望は多く頂きました。たしかに雑談によりユーザー同士やユーザーと開発者が身近になり、よりよいBotを作るための意思疎通ができたり、ユーザー同士のコミュニティがで

きるというメリットはあります。いっぽうで、トラブルへの対応や、トラブルを防ぐためのルールを管理するコストが大きくなることも予想できました。とれる時間はすべて開発にかけたいということを考え、雑談チャンネルは設けないことにしました。同様に、質問／要望チャンネルにおける雑談もご遠慮いただいています。

メンションではない投稿は、通知こそ送られませんが、設定によってはGuildアイコンの横に未読をあらわすマークが付きます（図16.3）。雑談でこの未読マークをつけないようにすることは、前述の「不要な通知は極力ユーザーに送らない」という目標にも適っていると考えています。

図16.3: 未読メッセージがあるときの通知表示

16.1.5　アップデート通知

サービスをアップデートした際には、ユーザーにアップデート通知を行います。アップデート通知では、追加した機能の用途や魅力がしっかり伝わるよう、表現方法を工夫しましょう。長過ぎるアップデート通知は読まれませんので、簡潔にわかりやすく表現しましょう。

アップデートは既存ユーザーに満足してもらうために行います。同時に、新規ユーザーを増やすチャンスでもあります。TwitterなどのSNSで魅力的なアップデート通知を投稿して、ユーザー増加を狙いましょう。投稿するタイミングですが、できれば、多くのユーザーがアクティブで、SNSを見ているユーザーの多いタイミングを狙います。筆者の経験上、一般的には11:00～13:00頃、16:00～18:00頃に情報拡散されやすい傾向があります。しかしDiscordユーザーにはゲーマーが多く、夜間活動型が多いためか、shovelのお知らせについては、21:00～23:00頃に活発に拡散されているようです。

16.1.5.1　Botからのアップデート通知は慎重に

Discord Botの場合、アップデート時、Botにアップデート内容を各Guildのテキストチャンネルに投稿させるという実装もアイデアとして浮かぶかもしれません。結論からいうと、これは避けたほうがよいでしょう。ユーザーの立場に立ってみると、「何も操作していないBotが突然テキストチャンネルにメッセージを投稿する」というのはあまり良い印象を与えません。Bot開発者は低頻度に抑えていると思っていても、ほとんど投稿のないGuildであれば「Botだけがたくさん喋っている」という印象になります。最悪の場合、Guildからキックされてしまうかもしれません。また、集中して真剣な会話をしているところにBotが急に割り込んでしまう可能性もあります。

どうしてもBotからアップデート内容を通知したいのであれば、なにかユーザーから命令を受けたとき、その結果と一緒に投稿したほうがよいでしょう。それも、しつこくならないようにするべきです。

16.2　ユーザーとどう関わるか考える

16.2.1　ユーザーサポートのラインを決める

ユーザーサポートをどのくらい行うかというのは、開発者自身で自由に決めてよいことです。まったく対応しないのでもかまいませんし、極力要望には応じるとしてもかまいません。どのようにしてもよいのですが、1点だけ推奨するならば、どのユーザーにも一貫性のある対応をするのがよいでしょう。あるユーザーには手厚くサポートし、要望を受け入れるが、他のユーザーにはろくに答えない、といった対応をしてはユーザーに不信感を与えます。もしそれがダイレクトメールなどの他人には見えない場であったとしても望ましくありません。

これは、ひどい要求をするユーザーにも丁寧に対応するべきという話ではありません。自分の中で、「ひどい要求は無視する」「このような質問には答えない」といったルールを定め、常にそのルールにしたがって対応するのがよい、ということです。

16.2.2　shovelのユーザーサポート

shovelでは、公式Guildで質問と要望を受け付けています。Twitterに関しては情報発信の場であるとし、フォローやリプライは行っていません。

16.2.2.1　質問への対応

説明書に書いてあるような基本的な使い方について尋ねられることも多いのですが、簡単に答えられる質問であれば、その場で答えてしまいます。もちろん説明書を読んでもらえたらありがたいのですが、説明書を読むのもなかなか大変なので、悩むようなら聞いてもらって構わないと考えています。いっぽうで、説明に時間がかかりすぎることであったり、何往復もやりとりが必要であるような質問やサポート依頼には原則対応しないことにしています。shovelの開発・運営に使える時間は限られており、一人のユーザーに長時間対応できないからです。

16.2.2.2　要望への対応

実装したいと思っていた機能についてユーザーに要望を頂くこともあれば、まったく思いもよらなかったことについて要望を頂き、なるほどと思って取り入れさせてもらうこともあります。たとえば、以下はユーザーの要望から追加した機能の例です。

・発言者の名前読み上げ
・絵文字読み上げ
・単語を「読まない」よう登録（NGワード）
・コマンド利用者制限機能

・spoiler（ネタバレ）を読まない

これは余談ですが、はじめshovel公式Guildには「要望」というチャンネルがありました。「要望」チャンネルには、「〜〜を〜〜する機能を追加してほしい」という具体的な機能についての要望を多くいただきました。しかし、これでは、そのユーザーが本質的に何に困っているのかがわからず、その機能を追加して問題解決につながるのかどうかがわかりませんでした。

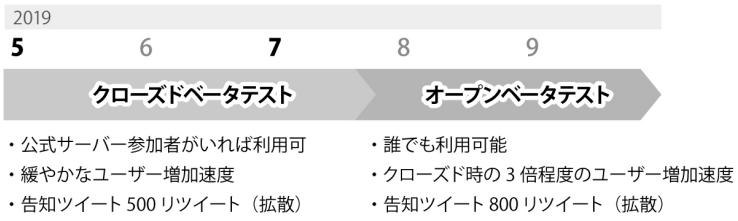
そこで、「困ったこと」というチャンネル名に変更しました。それ以来、ユーザーが困っていること自体について報告していただけるようになり、より本質に近い対応ができるようになりました。チャンネル名ひとつで書き手の表現が変わるというのは興味深いと感じました。

16.3　サービスを段階的に公開するメリットとデメリット

16.3.1　shovelの段階的サービス公開

ここでは、shovelがどのようにサービスを公開してきたかをお話しします。shovelは段階的にサービスの公開範囲を広げ、徐々にユーザーを増やしました（図16.4）。

図16.4: shovelの段階的サービス公開



16.3.2　まずはクローズドベータテストから

サービス開始から2か月とすこしの間、shovelはクローズドベータテストという形で公開しました。具体的には、特定の条件を満たすGuildでないと、shovelを利用できないというものです。特定の条件とは、「shovel公式Guildに参加しているメンバーがいる」というものでした。このような形態での公開とすることで、まずは開発者から確実に情報が届く利用者だけに利用してもらうことができました。

サービス開始時は、開発者である筆者のTwitterアカウントのフォロワー数は10に満たないという状態でした。shovelを使ってくれる人たちに情報が届くのかという不安がありました。ですが、さまざまな方に情報拡散を手伝っていただき、ありがたいことに1週間で500を超えるアカウントにリツイート（拡散）していただけることとなりました。

クローズドベータテストで、ユーザー数の増加の推移やBotの稼働状態とシステム負荷の相関を観察し、マシンスペックを増強したうえで、オープンベータテストに移行しました。

16.3.3　オープンベータテストへの移行

2019年7月中頃、shovelはオープンベータテストへと移行しました。このときから、shovel公式Guildに参加していなくても、誰でもshovelを自分のGuildへ招待可能になりました。オープンベータテストへの移行をしたあと、ユーザーの増加ペースが以前の3倍程度になりましたが、事前にマシンスペックを増強していたため、ユーザーの利用への影響はほぼありませんでした。

16.3.4　クローズドベータテストを経由する意味

オープンベータテストにして以降、ユーザーの増加ペースが3倍程度になったと述べました。この数字は、クローズドベータテストのときの「まず公式Guildに参加して、その後Botを招待する」という手順はわかりにくく、導入をあきらめていた潜在的なユーザーが多かったことを示しています。

これはある意味、使ってくれるはずだったユーザーを逃してしまったといえます。しかし、必ずしもこれがデメリットであるとは言い切れません。導入のハードルが下がるということは、多様なユーザーが、大人数で一気に流入することを意味します。Bot稼働の初期、足りない機能や不具合も多い中、多様なユーザーからの要望に対応するのは非常に負荷が高いものです。ですから、まずはその面倒な導入手順を踏んでくれるような、自分のBotに強い関心を持ってくれているユーザーだけに使ってもらい、ゆっくり確実に改修の初期段階を踏みます。これは前述のデメリットを補ってあまりあるものです。

目先の損失を惜しむより、そのとき利用してくれているユーザーに価値を提供しながら、ゆっくりとBotを育てて行くことに集中しましょう。爆発的な拡散の機会を逃しても、価値あるBotを継続的に安定して提供していれば、ユーザーは少しずつ、確実に増えていきます。shovelでは、広報ツイートの拡散が止まったあとも、利用Guildは指数関数的に増加しつづけています。おそらく、参加しているGuildでshovelが使われているのを見たユーザーが自分のGuildにもshovelを導入したり、shovelを使ったユーザーが友達に紹介したり、といった口コミ的な広がり方をしているのだと推測しています。

16.4　サービスの印象をデザインする

16.4.1　サービスの印象の重要性

サービスの印象とは、ここでは「サービスの名称（ネーミング）」と「サービスのビジュアル（アイコンやロゴ）」を指します。とくにDiscord Botの場合、ユーザーの目の前に表示されるのはアイコンと名前、あとは発言する内容のみです。アイコンと名前の重要性がおわかりいただけると思います。

サービスの印象は、テイクアウト飲食店のパッケージに例えることができます。パッケージデザインの善し悪しで飲食物の味は変わりませんが、買ったときの嬉しさや持ち歩いているときの気分に影響する可能性はあります。サービスもこれと同じことがいえます。

16.4.2　ネーミング

サービスの名称は、ターゲットとするユーザー層に親しまれやすいよう命名します。そのサービスがファミリー向けなのか、子供向けなのか、若者向けなのか、その層の気持ちになって最適な名前を考えましょう。名称は長くなってもかまいませんが、その場合覚えやすい・呼びやすい略称を付けましょう。

16.4.3　アイコン

サービスの内容、またはネーミングと一致したアイコンを作成します。shovelの場合、ネーミングに一致したアイコンです。唯一無二のビジュアルを付けるのもよいですが、親しまれやすい図案でわかりやすくそのサービスを表すのもよいでしょう。その場合、他者の権利を侵害しないよう十分に気を配りましょう。

16.4.4　注意点

16.4.4.1　他者の権利を侵害しないこと

アイコンやロゴ、イメージ画像を作る際には、他者の権利を侵害しないようにしましょう。他者が作成したイラストなどを勝手に使わないというのはもちろん、「フリー素材」と銘打たれていてもロゴを作る際の素材としては利用できないものもあります。規約をしっかり確認しましょう。

16.4.4.2　既存の類似ツールと被っていないか

名前やアイコンの印象が他ツールと被っていないかを確認します。唯一無二の名前というものはありませんが、同ジャンル・類似ツールで名前被りが発生していないかはしっかり調査しましょう。

16.4.4.3　サービスの品質が最優先

いくら覚えやすい名前を付けたとしても、その名前が悪い印象で覚えられてしまえば逆効果です。かっこいい名前と素敵なロゴがあっても、サービスとしての品質が悪くてはユーザーは離れてしまいます。サービスとしての品質を高く保つことがもっとも重要であることにかわりはありません。

16.4.5　shovelの印象デザイン

shovelのネーミングとアイコン作成は、年代性別をとわず、多くのDiscordユーザーに使ってほしいという思いで行いました。まずはネーミングです。毎日使う道具のようにBotを身近に感じてほしかったので、無機質でスマートな名前にしたいと考えました。しかし、ただスマートなだけでは面白くありません。そこで、「しゃべる」と「シャベル」をかけ、「shovel」という名前に決めました。

次にアイコンについてです。Discordのメインユーザー層はゲーマーであり、筆者自身もゲームが好きです。そこで、レトロゲームやMinecraftをイメージしたshovel独自のドット絵を作成し、これをアイコンに設定しました。

16.5　効果的な広報でBotを知ってもらおう

16.5.1　Twitterアカウントの設置

shovelは、公式Twitter（https://twitter.com/shovel_discord）で情報発信を行っています。shovelの情報発信の拠点は公式Guildです。なので、Twitterは、アップデートや障害の概要を手短に告知し、公式Guildに誘導するという補助的な役割です。

また、公式GuildはDiscord内にありますので、当然ながらDiscordに障害が発生したときは情報を発信できません。このようなときに、Twitterアカウントがあればdiscordの障害によりshovelが使えないことを広報できます。Discordに障害が発生しているときに読み上げが使えないことを周知しても意味がないのではと思われるかも知れませんが、障害の影響を受けているのが一部のGuildやBotだけという事態もよくあることですので、周知は必要です。

16.5.2　拡散される情報

shovelを紹介する広報ツイートは2020年8月現在、900近くリツイート（拡散）されています。この広報ツイートには、Discordのテキストチャンネルでshovelを呼び出し、複数のユーザーが登場し、名前読み上げや単語登録といった基本的な機能を紹介する1分弱の動画を添付しています。短いながら、shovelの魅力をしっかりと紹介したからこそ、これだけ拡散していただけたものと考えています。

16.6　Botに機能を追加する

16.6.1　機能を追加するかの検討

機能は、アイデアとして浮かんでくることもあれば、ユーザーからの要望という形で上がってくることもあります。もちろん、これらすべてをBotに実装できるわけではありません。機能を追加するかどうかは「Botの責務としてふさわしいか」「ユーザーの需要があるか」「開発者である自分が追加したいか」という3つの指標で検討します。

機能を追加することには注意が必要です。詳細は次の節でお話ししますが、削除するならはじめから付けないほうがよいこともあります。

16.6.1.1　Botの責務としてふさわしいか

まず、最低限の条件です。Discord Botは、ユーザーにできることなら基本的になんでもできます。そのため、「こんなことができてほしい」という要望にすべて応えていると、なんでも屋Botができ上がってしまいます。親しい友人向けのBotならなんでも屋Botになっても構わないでしょう。しかし大勢に使ってもらいたいならそれではいけません。機能が増えれば増えるほど操作方法は複雑になり、使うハードルは上がります。同じGuildにいるユーザーがそのBotを使っている様子を見た他のメンバーに魅力が伝わることはないでしょう。多機能すぎて何が売りのBotなのかわからないからです。そのBotの強みが何であるか、そのBotの責務は何なのかをつきつめて検討し、責務にふさわしくない機能は思い切って諦めましょう。新しいBotの開発をはじめてもいいかもしれませんね。

16.6.1.2　需要があるか

この観点で検討する際に注意しなければいけないのが、追加するよう要望をうけた機能が、そのまま需要がある機能であるとは言えない点です。ユーザーは自分でも認識していない要望を持っているものです。その潜在的な要望も含め、ユーザーの需要を見抜けるかというところがBot運営のおもしろいところです。「潜在的な要望なら見つけようがないじゃないか」というのはそのとおりで、その見積もりが正しいかどうかはリリースするまでわかりません（残念ながら大外れだったこともあります）。

そこで有効なのが、実装しようか迷っている機能についてユーザーに紹介し、欲しい機能はどれかというアンケートをとることです。しかし、個人的には投票数が少なかったからと言ってその機能を欲しいと言ってくれたユーザーの期待を裏切ることはしたくないため、あまり実施していません。ベータテストのごく初期に1度だけアンケートをとったことがありますが、選択肢にあった機能は最終的にすべて実装してしまいました。アンケートはあくまで、実装は決まっている機能について開発の優先順位付けをする参考に使うのがよいと考えています。

16.6.1.3　開発者である自分が追加したいか

　いろいろな指標がありますが、結局のところ、開発者である自分が欲しいと思うかどうかに帰結するのではないでしょうか？　場合によりますが、大抵の場合、開発者は自分が欲しいBotを作っています。使いたい機能を自由に実装できることは個人開発のメリットです。自分がほしいと思った機能は追加してよいといえるでしょう。もちろん、前述のふたつの指標との兼ね合いも検討しましょう。

16.7　Botから機能を削除する - 破壊的変更

16.7.1　破壊的変更は避けよう

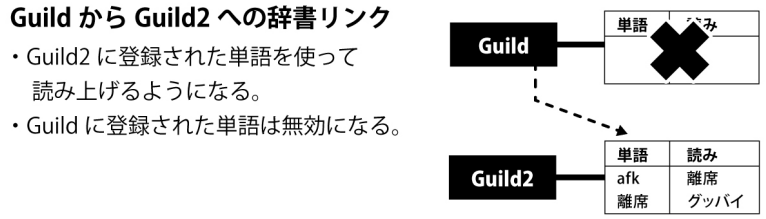
Botを運営していると、あまり使われていない機能を削除したり、コマンド名をより適切なものに修正したくなることがあります。このように、ユーザーのBotの使い方に影響を与えるようなことを、Botの破壊的変更と呼ぶことにします。Botの破壊的変更は、基本的には避けるべきことです。ここでは、機能の削除とコマンド名の変更について、実例とともに述べます。

16.7.2　機能の削除

ユーザーにとって、今まであった機能が削除されるというのは、もとからその機能がないことよりずっとマイナスです。利用ユーザーの数が少ない機能だったとしても、あるユーザーからすると、普段から頻繁に使っている機能かもしれません。ですので、機能を追加するときは、その機能の需要とその後の他機能に及ぼす影響などを鑑み、本当に追加してよいか慎重な検討が必要です。

shovelには、以前「辞書リンク機能」というものがありました。あるGuildで登録した単語たちを、他のGuildから参照して使用できるという機能です（図16.5）。

図16.5: 辞書リンク機能



しかし、この辞書リンク機能をいざリリースしたところ、1か月ほどたっても使用Guildが全体の0.1%にも満たず、コマンド自体ほぼ使われてもいない、超マイナー機能となってしまいました。

この機能は、単語登録内容というプライベートなデータを共有するという性質上、セキュリティを高めるための例外処理やチェックが多くなっています。そのうえ、読み上げ・単語登録というshovelのコアな部分に食い込んでいます。そのため、ちょっとした改修時に、本機能に影響がないかを気にする大きなコストがかかっていました。そこで、shovelがそのときベータテスト中であったこと、今後の実装コストと使用状況、類似機能である「import_word」「export_word」で代替できることなどを鑑みて、機能を削除することに決めまし

た。せっかく実装した機能を削除するのは残念だという気持ちもありましたが、なによりこの機能を使用してくれているGuildがあるのに削除してしまうというのが大変申し訳なく、機能の削除は二度としないようにしようと反省しました。

ちなみに、辞書リンク機能が使われなかった理由として思い当たる節はいろいろあります。中でも、使用できるユーザーが限られており（ふたつ以上のGuildのオーナーしか使えないコマンドでした）、使い方がややこしい（連鎖的な辞書リンクはできないなど）というのが何よりの原因だったと分析しています。繰り返しになりますが、それでも使用してくれていたユーザーには申し訳ないことをしてしまいました。

16.7.3　コマンド名の変更

続いて、コマンドの変更による破壊的変更についてです。shovelでは、辞書をファイルに書き出すコマンドと、ファイルから読み込むコマンドが存在します。これらのコマンドはもともと、書き出しは「word_list」、読み込みは「add_word_list」というコマンド名でした。しかし、「単語のリストを表示」という意味のコマンド文字列と、辞書のファイル書き出しという処理内容に齟齬があることに気づきました。また、add_word_listとadd_wordという似たコマンドが存在していることから、ユーザーが混乱していることもわかりました。これらの理由から、機能との対応がはっきりしている「export_word」「import_word」にコマンド文字列を変更することにしました。

コマンド変更も、ユーザーを混乱させるという意味では避けたほうがよいことですが、機能の削除ほど神経質にならなくてもかまいません。コマンド変更時には、公式Guildなどでアナウンスをするとともに、alias（別名）として古いコマンドを使用できるようにしておくというのも有効な手段です。discord.pyでは、どのaliasでコマンドが実行されたかが取得できます。コマンド変更後一定期間はaliasとして古いものを残しておき、古いコマンドが使われていた場合、機能は実行しつつも、このコマンドは名前が次のように変わりました、とユーザーに通知する処理を入れるのです。

16.8　動作詳細ログでアプリケーションを分析する

16.8.1　運営のヒントはユーザーの声だけではない

ユーザーとの接点は、直接のコミュニケーションだけではありません。ユーザーとサービスのインタラクションも接点のひとつです。サービス利用時のユーザーの振る舞いを動作詳細ログとして蓄積・分析することで、ユーザーが声にださない要望を見つけることができ、Botの改善につながることがあります。もちろんユーザーの会話内容など、プライバシーにかかわる内容については一切記録していません。

16.8.2　読み上げ文字数の分析でわかること

shovelの実例で説明します。「YomiageDetailLog」には、読み上げた投稿ひとつひとつについて、Guild ID、チャンネルID、投稿の文字数、合成音声のファイルサイズ等を保存しています。繰り返しになりますが、発言の内容などプライバシーにかかわる内容は保存していません。これらを集計することにより、「どのGuildが多くshovelを使っているか」や、「shovelが今日読み上げた文字数は何文字か」といったことがわかります。さらにこれらを集計・分析すると、「50文字以上の投稿は全体の1割程度だが、その1割の投稿により全体の30%程度の読み上げファイルサイズが占められている」といったことがわかります。この分析をもとにして、「読み上げ文字数は50文字前後を上限とすることでユーザー全体のサービスの満足度が上がるのではないか」などという仮説を立てることができ、Botの改善につながります。

さらに、読み上げ件数が増加傾向にある場合、その上昇傾向とメモリやCPUの逼迫状況を照らし合わせ、サーバースペックの増強の必要があるかを確認します。これにより、リソースが足りなくなる前に手を打つことができ、Botのダウンを未然に防ぐことができます。

16.8.3　コマンドログの分析でわかること

また、「CommandLog」には、ユーザーがshovelに対して指示した内容が保存されています。このログを分析することで、よく使われているコマンドや、逆に使われていないコマンドを見つけることができます。使われていないコマンドは、削除して機能を整理するか、使いやすいように改善する必要があるでしょう。

このように、動作詳細ログの分析を行うことで、ユーザー全体に利益のある仕様変更やサーバーのスペック変更を行うことができるのです。

第17章　Bot運営の金銭面

　ここでは、個人開発サービス運営を行ううえで出ていくお金と入ってくるお金について紹介します。また、shovelの金銭事情についても少し紹介します。筆者は会計・経理については素人です。公的な手続きを行う場合は本書を参考にするのではなくご自身で調査を行ってください。

17.1　サービス運営は儲かるのか？

17.1.1　Discord Botの傾向

Discord Botのユーザーは、当然Discordユーザーに限られます。また、Discordがゲーマーに人気であり、若年層が多いというユーザー分布の特徴があります。以上の条件から、ユーザー層を選ばないWebアプリケーション等と比較して、収益化しづらいという側面はあります。

ですが、Discordは全世界に2.5億人のユーザーを持つ巨大なプラットフォームです。Botの機能やジャンルを工夫すれば、Webアプリケーションより手軽に使ってもらいやすいぶん、ユーザーを増やしやすいかもしれません。たとえば、ユーザーに対してプレミアムプランを提供しているBotも存在します。

この章では、サービス運営の収入・支出を紹介しています。収益化を目指す方は参考にしてみてください。

17.1.2　収入を得るのはいいことばかりではない

個人開発サービスを有償提供することのメリットは、もちろん、資金を得ることによって運営に余裕が出ることです。では、デメリットはなんでしょうか。まず、ライセンスの問題です。無償利用、個人利用であれば無料で利用できるが、商用利用には制限のあるリソースは数多くあります。サービスに使っているライブラリーだけではなく、広報に用いているグラフィックについても注意が必要です。フォントや素材によっては、使用用途による制限があるかもしれません。

また、無料の個人開発サービスの運営は、自分の意志で行えることが楽しみのひとつです。ですが、万が一莫大な運営資金の提供を受けるようなことがあった場合、自分ひとりのサービスとして気ままに運営することができなくなってしまうかもしれません。

もちろん収入を得たからといって、すべてユーザーのいいなりになる必要はありません。それに、サービスの対価としてお金をもらっているだけであれば、契約にある以上の過剰な責任感を持つ必要はありません。ですが、お金をもらう以上はプロの運営者となるわけですから、無料で運営するのと同じ意識でいることはできません。いっぽうこれを逆手にとると、プロとして運営するという経験を得られるというのはある意味メリットかもしれません。

17.2　サービス運営で出ていくお金

サーバス運営に必要となるお金は、主に「インフラ利用料」「各種利用料」「人件費」です（図17.1）。

図17.1: 出て行くお金

17.2.1　インフラ利用料

一言でいうと、「サーバスを動かすためにかかるお金」です。自宅PCで運用している場合、電気代とインターネット回線費がそれにあたります。VPSで運営しているのであれば、その費用がインフラ利用料です。Herokuなどの無料枠で運用すれば無料です。どれくらいのコストがかかるのかというのは、サービスの利用者数や処理内容、要求する性能によってかなり幅があります。軽量なDiscord Botであれば、まずは月1000円程度で十分まかなえるでしょう。

17.2.2　各種利用料

サーバスを運営するのに必要なツールやアプリケーションに対してかかるお金です。「各種」と書いたように、作りたいサービスによってその内容は多岐にわたりますが、代表的なものを数点紹介します。

17.2.2.1　外部サービス利用料

監視ツールやストレージサービスなど、サーバーと連携して動作する外部サービスの利用料です。利用する外部サービスの種類によりますが、前述の「インフラ利用料」としてまとめて数えてしまってもよいでしょう。

17.2.2.2　ソフトウェアのライセンス料

有償ソフトウェアを利用してシステムを構築する場合、ライセンス料が必要です。個人利用と商用利用ではライセンス料が大幅に異なる場合が多いので、ソフトウェアを導入する際はライセンスの内容をしっかりと確認しましょう。

17.2.2.3　素材の利用料

システムによっては、画像や写真などの素材を有料で調達する必要があります。また、デザイナーやイラストレーターに有償でロゴやイラストを作成してもらう場合、それもサービスのための支出といえます。

17.2.3　人件費

開発者に支払われる賃金です。とはいえ、小規模なサービスであれば基本的に開発者はあなた一人でしょう。その場合お金としての支出はありませんが、人件費をゼロ円で見積もるのはやめましょう。それはつまりあなたの働いた価値をゼロと見積もるということです。あなたは何時間もかけてサービスをつくるという仕事をしたのです。サービスの開発や運営をしているぶんだけ、他の仕事で収入を得られるはずの時間が減っています。少なくともその分を計上すべきです。

もちろんすべての開発がビジネスを意識して行われるべきとはまったく思いませんが、開発にコスト意識を持つことは必要です。開発をテンポよくすすめサービスの運営を長く続けるためにも、人件費を意識することは有効な手段であるといえるでしょう。

17.3　サービス運営で入ってくるお金

サービスからお金を得るには、主に「有料プラン」「単発有償サポートプラン」「投げ銭」「広告収入」という4つの方法があります（図17.2）。

図17.2: 入ってくるお金

17.3.1　有料プラン

月額や年額でユーザーにお金を頂き、対価として機能を提供する方法です。対価には、「機能の追加」「制限の解除」があります。「機能の追加」は、有料プランユーザーのみが使える機能を用意するという方法です。いっぽう「制限の解除」は、無料ユーザーにも機能を提供し、期間内で一定の利用量に達するとその機能が制限されて使えなくなるところを、料金を支払えば制限なく使えるなどの方法です。

有料プランを採用するメリットとして、価値ある機能を提供さえできていれば、安定して収入を得続けられるということが挙げられます。また、ユーザーがサービスにどれくらいの価値を感じてくれているかが、お金という価値で可視化されるので、モチベーションにつながるでしょう。

有料ではあるものの、機能は何もかわらないという応援プランを採用することもできます。これについては投げ銭にあたるので、「投げ銭」の項をごらんください。

17.3.2　単発有償サポートプラン

サービスの導入や設定についての単発サポートを有償で提供する方法です。説明書だけでは使い方がわからないユーザーや、時間のないユーザーのために、導入や設定をサポートするかわりにお金を頂くという方法です。1時間以内の対応、というように時間を区切って単発で提供します。

単発有償サポートプランの存在は、そこから収入を得られる以外のメリットがあります。あるユーザーが開発者に対し、難しい内容のサポートを要求したときのことを考えてみましょう。その要求は少しの時間では解決できないと判断した開発者はユーザーの要求を断るでしょう。このとき、ユーザーは不親切だと感じ、開発

者もサポートを提供できなかったことを申し訳なく思うでしょう。結局、どちらも残念な思いを抱えることになります。ここで単発有償サポートプランがあればどうでしょうか。難しい要求があった時点で、開発者は単発有償サポートプランの存在を伝えます。「その要求は有償プランで対応できます」とだけ伝えればいいので、言葉を選ぶコストがありません。そこで、有償でもかまわないとユーザーが言えば、開発者は収入を得ることができますし、ユーザー側もお金を払って気兼ねなくサポートを受けることができます。

もちろん時間に余裕があればこのようなプランは必要ないかもしれません。金目当てか、というような批判もあるでしょう。しかし、ひとりのユーザーに時間をかけることでサービスの開発に遅れが生じるとしたら、それはすべてのユーザーの損失となってしまいます。厚意でサポートをするのは自由ですが、個人開発という時間が限られている中で自分の時間をどう使うかということには常に敏感でいる必要があります。

17.3.3　投げ銭

投げ銭とは、ユーザーに金銭の対価を提供せず、ただ応援のためにお金を頂くことをいいます。投げ銭の窓口として、PayPalやBOOTHを利用している開発者を見たことがあるかもしれません。投げ銭はあくまでいち開発者として応援を受ける行動であり、直接的にサービスの価値をビジネスに利用しているわけではありません。そのため、前述したような商用利用に特別な条件が課されているライセンスの制約を回避できる場合もあります。もちろん投げ銭としてでも収入を得ているのであれば、商用利用とみなすというライセンスも多いため、必ず自身でよく確認しましょう。

直接的な金銭の受け取りではありませんが、Amazonほしいものリストから贈り物を募集している開発者もよく見かけますね。

17.3.4　広告収入

ユーザーに直接課金するのではなく、Webサイトなどに貼る広告を通じて間接的に収入を得る方法です。ただし、Discord Botについては注意が必要です。Discord APIを利用した広告配信は2020年8月現在利用規約で禁止されており、Botから広告を投稿することは規約違反にあたります。広告を配信するのであれば、説明書や紹介ブログなど間接的なものをプラットフォームにする方法があります。

17.4　shovelのお金事情

17.4.1　支出

まずサーバー代についてお話しします。shovelは約3万のGuildに導入されており、音声合成をサーバー内で行いボイスチャンネルに流すという、処理内容もネットワーク使用量もやや負荷の高いBotであるといえます。shovelは、Kagoya VPSのKVMプラン（8GB）を利用しています。年間約10万円ほどです。監視ツールや関連アプリケーションはすべて無料のものを使っています。あとは開発者である筆者自身の人件費です。

17.4.2　収入

shovelには有料プランはないので、shovel自体から得られる収入は0円です。shovelの説明書を掲載しているブログからの広告収入が少しありますが、shovelのみで収益化できているという状態ではありません。

17.4.3　shovelの収益化

「使用料を払いたい」「有料プランを用意してほしい」という声を多く頂いており、少額とはいえ金銭的な持ち出しがあるのも事実なので、投げ銭インターフェイスとしてpixivFANBOXを利用しています。また、読み上げ速度や品質を向上させた有料プランのshovelを提供することも検討中です。

おわりに

あとがき - 個人開発のよろこび

すべての責任と裁量が自分にある、それが個人開発です。これまで趣味程度の小さなアプリケーションしか開発したことがなかった筆者にとって、shovelを広く世間に公開するというのはとても大きなチャレンジでした。そもそも、友人の強いすすめがなければ、やろうとも思わなかったでしょう。結局、公開すると決意してからリリースするまでに8か月もかかってしまいました。

ですが、今となっては、本当にやってよかったという思いしかありません。shovel（の前身であるBot）を作ったときに筆者自身が感じた「便利！」という気持ちを、使ってくれたユーザーが共有してくれたであろうこと。ユーザーのDiscordでのコミュニケーションが今までよりもっと楽しくなったであろうこと。今も接続中のGuildの数を見ながら、その数字の向こうにいるユーザーのことを想像し、本当に嬉しい気持ちになります。もちろん、身近な人に喜ばれるのも嬉しいことです。ですが、50万人という普通に生きているだけでは手が届かない人数に対し、価値あるBotを提供できたというのは、自分にとって非常に大きな出来事でした。

いま何かサービスを作りたいと思っている人や、作ったサービスを広く公開することに二の足を踏んでいる方がいたら、ぜひとも勇気を出して、一歩踏み出してみてください。ちょっとした苦労はあるでしょうが、そのすべては糧になります。そして、それ以上の楽しみ、喜びを得ることができるでしょう。この本がその一歩を踏み出す手助けをできるとしたら、こんなにうれしいことはありません。

謝辞

この本を書くにあたって、たくさんの方にお世話になりました。その内容についてはここでは書き切れないので、名前だけ挙げさせていただきます。本当にありがとうございました。まず、お忙しい中査読を引き受けてくださったのは次の13名の方々です。感謝にたえません。おかげさまで心強い思いでこの本を出せました。

（あいうえお、abc順）

・石本敦夫 様（@atsuoishimoto）
・かんだ 様
・香田ユウ 様（@aromarious）, https://aromarious.com/
・さちえ 様（@curry_solodev）, https://triokini.com/
・するめ 様
・丹内駿 様（@1ntegrale9）
・なちゅ。様（@itacchiku）, https://note.mu/itacchiku
・ひよこ 様
・aiotter 様, https://github.com/aiotter
・anpontan 様
・Devon R 様, https://github.com/Gorialis
・mxkitaura 様
・Tomo 様

本の執筆にあたっていろいろと支えてくださったのは次の方々です。ありがとうございました。

・父、母、友人M、犬

また、shovelユーザーの皆様がいなければこの本を書くことはありませんでした。拙い開発と運営のBotにもかかわらず、いつもたくさん利用していただき、本当にありがたく思っています。これからもshovelをよろしくお願い致します。

最後に、shovelを公開することを強く勧めてくれたうえに、私がshovelの開発・運営で壁に当たるたび最低限の的確なヒントを出して解決へと導き、本書の執筆にも多くのアドバイスと手助けをして下さった香田ユウ氏に深く感謝の意を表します。

2020年8月　北浦 望

著者紹介

北浦 望・cod（きたうら のぞみ）

ソフトウェアエンジニア。プログラミング歴は業務含め7年ほど。趣味で開発・運営する日本語読み上げDiscord Bot「shovel」はサービス公開から1年間で50万ユーザーを突破。趣味としてグラフィックデザインにも取り組む。好きな寿司ネタはサーモンと炙りえんがわ。Twitter ID: @cod_sushi /技術中心ブログ: https://cod-sushi.com

◎本書スタッフ

アートディレクター/装丁：岡田章志＋GY

編集協力：深水 央

デジタル編集：栗原 翔

技術の泉シリーズ・刊行によせて

技術者の知見のアウトプットである技術同人誌は、急速に認知度を高めています。インプレスR&Dは国内最大級の即売会「技術書典」（https://techbookfest.org/）で頒布された技術同人誌を底本とした商業書籍を2016年より刊行し、これらを中心とした『技術書典シリーズ』を展開してきました。2019年4月、より幅広い技術同人誌を対象とし、最新の知見を発信するために『技術の泉シリーズ』へリニューアルしました。今後は「技術書典」をはじめとした各種即売会や、勉強会・LT会などで頒布された技術同人誌を底本とした商業書籍を刊行し、技術同人誌の普及と発展に貢献することを目指します。エンジニアの"知の結晶"である技術同人誌の世界に、より多くの方が触れていただくきっかけになれば幸いです。

株式会社インプレスR&D
技術の泉シリーズ　編集長　山城 敬

●お断り

掲載したURLは2020年7月1日現在のものです。サイトの都合で変更されることがあります。また、電子版ではURLにハイパーリンクを設定していますが、端末やビューアー、リンク先のファイルタイプによっては表示されないことがあります。あらかじめご了承ください。

●本書の内容についてのお問い合わせ先

株式会社インプレスR&D　メール窓口
np-info@impress.co.jp

件名に「『本書名』問い合わせ係」と明記してお送りください。
電話やFAX、郵便でのご質問にはお答えできません。返信までには、しばらくお時間をいただく場合があります。
なお、本書の範囲を超えるご質問にはお答えしかねますので、あらかじめご了承ください。
また、本書の内容についてはNextPublishingオフィシャルWebサイトにて情報を公開しております。
https://nextpublishing.jp/

技術の泉シリーズ

個人開発サービス運営実践入門
50万人が使うDiscord Bot「shovel」の舞台裏

2020年9月18日　初版発行Ver.1.0（リフロー版）

著　者　北浦 望
編集人　山城 敬
企画・編集　合同会社技術の泉出版
発行人　井芹 昌信
発　行　株式会社インプレスR&D
〒101-0051
東京都千代田区神田神保町一丁目105番地
https://nextpublishing.jp/

ISBN978-4-8443-7864-8

NextPublishing®

◉本書はNextPublishingメソッドによって発行されています。
NextPublishingメソッドは株式会社インプレスR&Dが開発した、電子書籍と印刷書籍を同時発行できるデジタルファースト型の新出版方式です。 https://nextpublishing.jp/