Slack Botと継続モナド

Slack Bot with the Continuation Monad

吉村 優
Hikaru Yoshimura

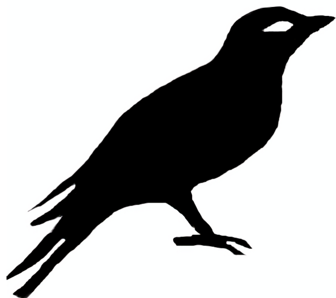
yyu@mental.poker

June 13, 2018
(Git Commit ID: 56fb0b7)

目次

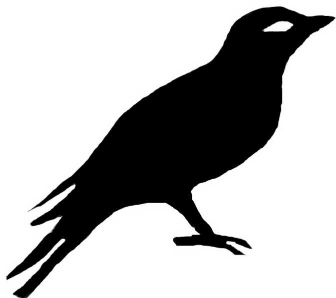
1. 自己紹介
2. Slack Bot とは？
3. 例となる機能
4. コールバック渡しスタイル
5. 継続モナド
6. まとめ

自己紹介



Twitter	@_yyu_
Qiita	yyu
GitHub	y-yu

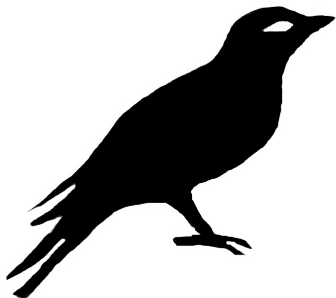
自己紹介



- 筑波大学 情報科学類卒（学士）

Twitter	@_yyu_
Qiita	yyu
GitHub	y-yu

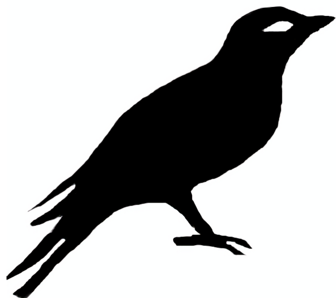
自己紹介



- 筑波大学 情報科学類卒（学士）
- プログラム論理研究室

Twitter	@_yyu_
Qiita	yyu
GitHub	y-yu

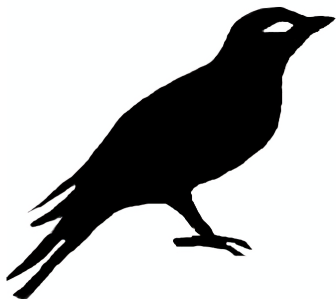
自己紹介



- 筑波大学 情報科学類卒（学士）
- プログラム論理研究室
- 最近 Slack Bot をつくったら楽しかった！

Twitter	@_yyu_
Qiita	yyu
GitHub	y-yu

自己紹介



Twitter	@_yyu_
Qiita	yyu
GitHub	y-yu

- 筑波大学 情報科学類卒（学士）
- プログラム論理研究室
- 最近 Slack Bot をつくったら楽しかった！
- 今回はその話をします

Slack Bot とは？

Slack Bot とは？

Slack Bot

Slack でメンションしたりすると、なんかしてくれるプログラム

Slack Bot とは？

Slack Bot

Slack でメンションしたりすると、なんかしてくれるプログラム

- たとえば

Slack Botとは？

Slack Bot

Slack でメンションしたりすると、なんかしてくれるプログラム

- たとえば
  - 投票

Slack Bot とは？

Slack Bot

Slack でメンションしたりすると、なんかしてくれるプログラム

- たとえば
  - 投票
  - TODO リスト管理

Slack Bot とは？

Slack Bot

Slack でメンションしたりすると、なんかしてくれるプログラム

- たとえば
  - 投票
  - TODO リスト管理
  - 検証端末管理

例となる機能

例となる機能

> “`Hello`” という投稿に対して、“`World`” という投稿をする

ナイーブな実装

ナイーブな実装

- たとえばすごく簡単にはこんな実装ができる

```
def helloWorldBot(message: Message): Unit = {
  if (message.text == "Hello") {
    sendMessage(message.channel, "World")
  } else {
    ()
  }
}
```

ナイーブな実装

- たとえばすごく簡単にはこんな実装ができる

```
def helloWorldBot(message: Message): Unit = {
  if (message.text == "Hello") {
    sendMessage(message.channel, "World")
  } else {
    ()
  }
}
```

- ここにログ出力を追加したとする

ナイーブな実装

- たとえばすごく簡単にはこんな実装ができる

```
def helloWorldBot(message: Message): Unit = {
  if (message.text == "Hello") {
    sendMessage(message.channel, "World")
  } else {
    ()
  }
}
```

- ここにログ出力を追加したとする

```
def helloWorldBot(message: Message): Unit = {
  if (message.text == "Hello") {
    sendMessage(message.channel, "World")
    logger.info("Success!")
  } else {
    ()
  }
}
```

ナイーブな実装

```
def helloWorldBot(message: Message): Unit = {
  if (message.text == "Hello") {
    sendMessage(message.channel, "World")
    logger.info("Success!")
  } else {
    ()
  }
}
```

ナイーブな実装

```
def helloWorldBot(message: Message): Unit = {
  if (message.text == "Hello") {
    sendMessage(message.channel, "World")
    logger.info("Success!")
  } else {
    ()
  }
}
```

- あれ？ このままだとログを常に出力する感じになったぞ

ナイーブな実装

```scala
def helloWorldBot(message: Message): Unit = {
  if (message.text == "Hello") {
    sendMessage(message.channel, "World")
    logger.info("Success!")
  } else {
    ()
  }
}
```

- あれ？ このままだとログを常に出力する感じになったぞ
  - まあ、`helloWorldBotWithoutLog`とかを作ってごまかすという手もあるが……

ナイーブな実装

```
def helloWorldBot(message: Message): Unit = {
  if (message.text == "Hello") {
    sendMessage(message.channel, "World")
    logger.info("Success!")
  } else {
    ()
  }
}
```

- あれ？ このままだとログを常に出力する感じになったぞ
  - まあ、helloWorldBotWithoutLogとかを作ってごまかすという手もあるが……
  - とはいえ、もっと機能が増えたらその手の関数が増えまくってめちゃくちゃになるのは容易に想像できる

ナイーブな実装

```
def helloWorldBot(message: Message): Unit = {
  if (message.text == "Hello") {
    sendMessage(message.channel, "World")
    logger.info("Success!")
  } else {
    ()
  }
}
```

- あれ？ このままだとログを常に出力する感じになったぞ
  - まあ、helloWorldBotWithoutLogとかを作ってごまかすという手もあるが……
  - とはいえ、もっと機能が増えたらその手の関数が増えまくってめちゃくちゃになるのは容易に想像できる
- 関数に機能を追加していくと**再利用の粒度**が粗くなる！

コールバック渡しスタイル

コールバック渡しスタイル

- コールバック関数を渡せばいいのでは？

コールバック渡しスタイル

- コールバック関数を渡せばいいのでは？

```
def helloWorldBotCallback(
  message: Message,
  callback: String => Unit
): Unit = {
  if (message.text == "Hello") {
    sendMessage(message.channel, "World")
    callback("Success!")
  } else {
    ()
  }
}
```

コールバック渡しスタイル

- コールバック関数を渡せばいいのでは？

```
def helloWorldBotCallback(
  message: Message,
  callback: String => Unit
): Unit = {
  if (message.text == "Hello") {
    sendMessage(message.channel, "World")
    callback("Success!")
  } else {
    ()
  }
}
```

- ログ出力するときはコールバックにロガーを突っ込む

コールバック渡しスタイル

- コールバック関数を渡せばいいのでは？

```
def helloWorldBotCallback(
  message: Message,
  callback: String => Unit
): Unit = {
  if (message.text == "Hello") {
    sendMessage(message.channel, "World")
    callback("Success!")
  } else {
    ()
  }
}
```

- ログ出力するときはコールバックにロガーを突っ込む

```
helloWorldBotCallback(message, logger.info)
```

コールバック渡しスタイル

- コールバック関数を渡せばいいのでは？

```
def helloWorldBotCallback(
  message: Message,
  callback: String => Unit
): Unit = {
  if (message.text == "Hello") {
    sendMessage(message.channel, "World")
    callback("Success!")
  } else {
    ()
  }
}
```

- ログ出力するときはコールバックにロガーを突っ込む

```
helloWorldBotCallback(message, logger.info)
```

- ログ出力しないときは何もしない関数を突っ込む

コールバック渡しスタイル

- コールバック関数を渡せばいいのでは？

```
def helloWorldBotCallback(
  message: Message,
  callback: String => Unit
): Unit = {
  if (message.text == "Hello") {
    sendMessage(message.channel, "World")
    callback("Success!")
  } else {
    ()
  }
}
```

- ログ出力するときはコールバックにロガーを突っ込む

```
helloWorldBotCallback(message, logger.info)
```

- ログ出力しないときは何もしない関数を突っ込む

```
helloWorldBotCallback(message, _ => ())
```

コールバック渡しスタイル

- これで解決したか？

コールバック渡しスタイル

- これで解決したか？
- 冷静に考えたら、この関数は 3 つのことをやっている

コールバック渡しスタイル

- これで解決したか？
- 冷静に考えたら、この関数は 3 つのことをやっている
    1. メッセージが “`Hello`” と等しいか判定する

コールバック渡しスタイル

- これで解決したか？
- 冷静に考えたら、この関数は 3 つのことをやっている
  1. メッセージが “`Hello`” と等しいか判定する
  2. “`World`” というテキストの投稿する

コールバック渡しスタイル

- これで解決したか？
- 冷静に考えたら、この関数は 3 つのことをやっている
  1. メッセージが “`Hello`” と等しいか判定する
  2. “`World`” というテキストの投稿する
  3. ログを出力する

コールバック渡しスタイル

- これで解決したか？
- 冷静に考えたら、この関数は 3 つのことをやっている
  1. メッセージが “`Hello`” と等しいか判定する
  2. “`World`” というテキストの投稿する
  3. ログを出力する
- このうち、3はコールバックにした

コールバック渡しスタイル

- これで解決したか？
- 冷静に考えたら、この関数は 3 つのことをやっている
  1. メッセージが “`Hello`” と等しいか判定する
  2. “`World`” というテキストの投稿する
  3. ログを出力する
- このうち、③はコールバックにした
- なら、②もコールバックでよくない？

コールバック渡しスタイル

- まずは `sendMessage`関数をコールバック化する

コールバック渡しスタイル

- まずは sendMessage関数をコールバック化する

```
def sendWorldCallback(
  channel: String,
  callback: Unit => Unit
): Unit = {
  sendMessage(channel, "World")
  callback(())
}
```

コールバック渡しスタイル

- そして判定しかしない `helloCallback`を作る

コールバック渡しスタイル

- そして判定しかしない helloCallbackを作る

```
def helloCallback(
  message: Message,
  callback: String => Unit
): Unit = {
  if (message.text == "Hello") {
    callback("Success!")
  } else {
    ()
  }
}
```

コールバック渡しスタイル

- そして判定しかしない `helloCallback`を作る

```
def helloCallback(
  message: Message,
  callback: String => Unit
): Unit = {
  if (message.text == "Hello") {
    callback("Success!")
  } else {
    ()
  }
}
```

- メッセージが `Hello` でないときはコールバックを実行しない

コールバック渡しスタイル

- そして判定しかしない `helloCallback`を作る

```
def helloCallback(
  message: Message,
  callback: String => Unit
): Unit = {
  if (message.text == "Hello") {
    callback("Success!")
  } else {
    ()
  }
}
```

- メッセージが `Hello` でないときはコールバックを実行しない
- つまり、後続の処理を実行するかどうかはその前の関数のたなごころ次第だ！

コールバック渡しスタイル

- そして判定しかしない `helloCallback`を作る

```
def helloCallback(
  message: Message,
  callback: String => Unit
): Unit = {
  if (message.text == "Hello") {
    callback("Success!")
  } else {
    ()
  }
}
```

- メッセージが `Hello` でないときはコールバックを実行しない
- つまり、後続の処理を実行するかどうかはその前の関数のたなごころ次第だ！
  - 関数を並べただけでは「前の関数が後続の関数を実行するか決める」といった制御はフラグを無理やり使わなければできない

コールバック渡しスタイル

- そうしたら全てがコールバックになるぞ！

コールバック渡しスタイル

- そうしたら全てがコールバックになるぞ！

```
def helloWorldBot(message: Message): Unit = {
  helloCallback(message, out =>
    sendWorldCallback(
      message.channel,
      _ => logger.info(out)
    )
  )
}
```

コールバック渡しスタイル

- そうしたら全てがコールバックになるぞ！

```
def helloWorldBot(message: Message): Unit = {
  helloCallback(message, out =>
    sendWorldCallback(
      message.channel,
      _ => logger.info(out)
    )
  )
}
```

- イノセントな再利用性、スパースな結合

コールバック渡しスタイル

- そうしたら全てがコールバックになるぞ！

```
def helloWorldBot(message: Message): Unit = {
  helloCallback(message, out =>
    sendWorldCallback(
      message.channel,
      _ => logger.info(out)
    )
  )
}
```

- イノセントな再利用性、スパースな結合
- これは素晴しい Slack Bot になりそうだ

コールバック渡しスタイル

- そうしたら全てがコールバックになるぞ！

```
def helloWorldBot(message: Message): Unit = {
  helloCallback(message, out =>
    sendWorldCallback(
      message.channel,
      _ => logger.info(out)
    )
  )
}
```

- イノセントな再利用性、スパースな結合
- これは素晴しい Slack Bot になりそうだ
  - プログラムがここまで美しいなら、きっとその振舞いも素晴しいに違いない！

コールバック渡しスタイル

- たしかに再利用の粒度は向上した

コールバック渡しスタイル

- たしかに再利用の粒度は向上した
  - もともと 3 つの機能が押し込んであった関数をそれぞれ分離した

コールバック渡しスタイル

- たしかに再利用の粒度は向上した
  - もともと 3 つの機能が押し込んであった関数をそれぞれ分離した
- でも、これはちょっと書くのが面倒な気がするぞ……

コールバック渡しスタイル

- たしかに再利用の粒度は向上した
  - もともと 3 つの機能が押し込んであった関数をそれぞれ分離した
- でも、これはちょっと書くのが面倒な気がするぞ……
- **継続モナド（Continuation Monad）**に隠蔽しよう！
  - 継続 = コールバック

継続モナド

継続モナド

- 定義はすごくシンプル

継続モナド

- 定義はすごくシンプル

継続モナド（Continuation Monad）

```scala
case class Cont[R, A](run: (A => R) => R) {
  def map[B](f: A => B): Cont[R, B] =
    Cont(k => run(a => k(f(a))))
  def flatMap[B](f: A => Cont[R, B]): Cont[R, B] =
    Cont(k => run(a => f(a).run(k)))
}
```

継続モナド

- 定義はすごくシンプル

継続モナド（Continuation Monad）

```
case class Cont[R, A](run: (A => R) => R) {
  def map[B](f: A => B): Cont[R, B] =
    Cont(k => run(a => k(f(a))))
  def flatMap[B](f: A => Cont[R, B]): Cont[R, B] =
    Cont(k => run(a => f(a).run(k)))
}
```

- これを今すぐ理解するのは無理かもしれないけど、こういうのもあるということは覚えておいていい

継続モナド

- 定義はすごくシンプル

継続モナド（Continuation Monad）

```
case class Cont[R, A](run: (A => R) => R) {
  def map[B](f: A => B): Cont[R, B] =
    Cont(k => run(a => k(f(a))))
  def flatMap[B](f: A => Cont[R, B]): Cont[R, B] =
    Cont(k => run(a => f(a).run(k)))
}
```

- これを今すぐ理解するのは無理かもしれないけど、こういうのもあるということは覚えておいていい
- 継続の界隈ではコールバック関数に k という変数を用いるのが一般的

継続モナド

- あとは気合で書き換えるだけ

継続モナド

- あとは気合で書き換えるだけ

```
def helloCont(message: Message): Cont[Unit, String] =
  Cont { k =>
    if (message.text == "Hello") {
      k("Success!")
    } else {
      ()
    }
  }

def sendWorldCont(channel: String): Cont[Unit, Unit] =
  Cont { k =>
    sendMessage(channel, "World")
    k(())
  }

def loggerCont(message: String): Cont[Unit, Unit] =
  Cont { k =>
    logger.info(message)
    k(())
  }
```

継続モナド

- そして `for`式で合成すれば OK

継続モナド

- そして `for`式で合成すれば OK

```
def helloWorldBot(message: Message): Unit = {
  val cont = for {
    logMsg <- helloCont(message)
    _ <- sendMessageCont(message.channel)
    _ <- loggerCont(logMsg)
  } yield ()

  cont.run(x => x)
}
```

継続モナド

- そして `for`式で合成すれば OK

```
def helloWorldBot(message: Message): Unit = {
  val cont = for {
    logMsg <- helloCont(message)
    _ <- sendMessageCont(message.channel)
    _ <- loggerCont(logMsg)
  } yield ()

  cont.run(x => x)
}
```

- コールバックを渡しまくってたバージョンより洗練された！

まとめ

まとめ

- 継続を使うと再利用の粒度をもっと細かくできる

まとめ

- 継続を使うと再利用の粒度をもっと細かくできる
- 生の継続渡しはちょっと使いづらい……

まとめ

- 継続を使うと再利用の粒度をもっと細かくできる
- 生の継続渡しはちょっと使いづらい……
- 継続モナドを使うと `for`式で綺麗に書ける

まとめ

- 継続を使うと再利用の粒度をもっと細かくできる
- 生の継続渡しはちょっと使いづらい……
- 継続モナドを使うと `for`式で綺麗に書ける
- こいつを使えば最強の Slack Bot が作れるに違いない！

まとめ

- 継続を使うと再利用の粒度をもっと細かくできる
- 生の継続渡しはちょっと使いづらい……
- 継続モナドを使うと `for`式で綺麗に書ける
- こいつを使えば最強の Slack Bot が作れるに違いない！
    - https://github.com/y-yu/slackcont

まとめ

- 継続を使うと再利用の粒度をもっと細かくできる
- 生の継続渡しはちょっと使いづらい……
- 継続モナドを使うと `for`式で綺麗に書ける
- こいつを使えば最強の Slack Bot が作れるに違いない！
  - https://github.com/y-yu/slackcont
  - ここに実装があるぞ

Thank you for your attention!