Rubyプログラミングで絶対にやってはいけない25のこと

尺一麟 著

目次

【注意事項】

・本書の内容は、その正確性や完全性を保証するものではありません。

・本書の内容を実践する場合は、自己責任で行ってください。

・著者や出版社は、いかなる損害についても責任を負いかねます。

・小説やフィクションというわけではありませんが、ある種のフィクションのような感覚も常に持ちつつ、どこか俯瞰的な視点を持って、学習したり、参考程度にお楽しみいただければ幸いです。

・本書は、完全にではございませんが、AIを使用しながら書いていますので、過度なクオリティや読みやすい文章の期待はしないでください。

ただ、そういう意味では、AIライティングにおける1つの参考例としてお読みいただくのも1つの楽しみ方かなと個人的には思います。

以上のことを踏まえた上で、もし宜しければお読みいただければ幸いです。

少しでも参考になる部分がございましたら幸いです。

また、Kindleを読む際は、読む本について、学びたいことを明確にし、それに集中して読むことによって学習効果が高まることが期待できます。

加えて、Kindleでは、ハイライトとメモを使うことができますので、大事だと思ったところや自分なりの理解をメモすることによって、あとで見返すことができて、より深い理解に繋がりますので、もし宜しければお試しください。

期間限定プレゼントのお知らせ

大事な内容なので、冒頭でお知らせを失礼いたします。（お知らせといっても、お金がかかる話ではございません。）

現在、下記のLINE公式アカウントを友達追加していただいた場合に、「Pythonプログラミングをする時に絶対に注意したほうがいい20のこと」（40ページのPDF）を無料プレゼントしています。

これは、Kindleとして出版予定だったものですが、LINE公式アカウントを友達追加してくれた方への感謝の気持ちとして無料でプレゼントすることにしました。

とても大事な内容が書かれていますので、ぜひ参考にしていただければ幸いです。

こういう表紙でリリース予定でした。

URLから友達追加する場合はこちらからできます。

https://lin.ee/HNLJYFR

こちらは、QRコードです。

また、IDで検索する場合は、

@334upsmr

です。

※予告なく締め切って有料のKindleとして出版する可能性もある内容ですので、ぜひ受け取れる内に受け取っておいていただくことをおすすめします。

前置きを失礼いたしました。それでは、本編の内容に移らせていただきます。

絶対にやってはいけないこと1

未宣言の変数を使用する。

未宣言の変数を使用すると、プログラムが正しく動作しない可能性があります。未宣言の変数は、その名前を持つメモリ領域が割り当てられていないため、予期せぬ結果を引き起こすことがあります。また、未宣言の変数を使用することはコードの可読性を低下させ、後のメンテナンスやデバッグ時に問題を引き起こす可能性もあります。

そのため、Rubyプログラミングでは必ず変数を宣言し、適切な初期値を与えることが重要です。変数を使う前に宣言し、そのスコープ内で正

しく使用することで、予期せぬエラーを避けることができます。

例えば、以下のコードは未宣言の変数を使用しています。

```ruby
x = 10
y = x + z
puts y
```

この場合、変数`z`は宣言されていないため、エラーが発生します。

```
```

undefined local variable or method `z' for main:Object (NameErr

or)

```

このエラーは、未宣言の変数を使用したことによるものです。このようなエラーを避けるためには、変数を使用する前に必ず宣言し、適切な値を与えるように注意しましょう。

絶対に未宣言の変数を使用しないようにすることは、Rubyプログラミングで重要なポイントです。宣言されていない変数を使用することは予期せぬバグを引き起こす可能性があるため、常に適切な変数の宣言と初期化を心掛けましょう。
```

絶対にやってはいけないこと2

他のプログラム言語からコピーしたコードをそのまま貼り付ける。

Rubyプログラミングを始めたばかりの人々にとって、他のプログラム言語からコードをコピーしてくることは誘惑となることでしょう。しかし、これは絶対に避けるべき行為です。なぜなら、他のプログラム言語のコードは、その言語特有の構文やルールに基づいているからです。

Rubyは独自の特徴と構文を持ち、他のプログラム言語とは異なる場合があります。そのため、他の言語からコードをコピーしてきても、そのままでは正しく動作しないことがあります。例えば、他の言語ではセミ

コロンで文を終了する必要があるかもしれませんが、Rubyでは不要です。そのような微妙な違いが、コピーしたコードの動作を妨げる可能性があります。

また、コピーしたコードは他の言語での仕様に基づいているため、Rubyの最適な書き方ではないかもしれません。Rubyの特性を活かしたコードを書くためには、適切な構文やイディオムを学ぶ必要があります。他の言語からのコードのコピーは、その学習過程を妨げる可能性があります。

したがって、Rubyプログラミングを学ぶ際は、他のプログラム言語からのコードコピーを避けましょう。代わりに、Rubyの公式ドキュメントや信頼できる学習リソースを活用し、Ruby固有の構文やベストプラ

クティスを学んでください。自分自身でコードを書き、エラーやバグを修正することで、より良いプログラマーになることができます。

絶対にやってはいけないこと3

ゴルフスクリプトのように最小の文字数で書こうとする。

ゴルフスクリプトのように最小の文字数でコードを書こうとすることは、Rubyプログラミングでは絶対にやってはいけないことです。

Rubyは読みやすさや保守性に重点を置いたプログラミング言語であり、可読性の高いコードが推奨されています。最小の文字数で書くと、

コードが複雑で理解しづらくなる可能性があるため、他の人がプログラムを理解しにくくなります。

また、ゴルフスクリプトを使うと、明確な変数名や繰り返し構造の使用が制限されるため、プログラムの目的や機能が不明瞭になる可能性もあります。これにより、将来の保守や修正が困難になる可能性もあります。

ゴルフスクリプトのように最小の文字数で書くことは、危険であり、可読性や保守性の観点からも避けるべきです。プログラムの品質を向上させるためには、明確な変数名やコメントの使用、コードのモジュール化など、良いプラクティスを守ることが重要です。

絶対にやってはいけないこと4

大規模なアプリケーションを1つのファイルに詰め込む。

大規模なアプリケーションを1つのファイルに詰め込むことは、Rubyプログラミングにおいて絶対にやってはいけないことです。

大規模なアプリケーションを1つのファイルに詰め込むことは、可読性や保守性の面で大きな問題を引き起こします。プログラムが巨大になるほど、コードの理解や修正が難しくなります。また、機能を独立して切り分けることができないため、変更や拡張が非常に困難になります。

代わりに、アプリケーションを論理的にモジュールに分割し、それぞれのモジュールごとにファイルを作成することが重要です。これにより、関連する機能をグループ化し、可読性を向上させることができます。また、各モジュールを独立してテストすることも可能になります。

さらに、関数やクラスの再利用性を高めるために、適切なモジュール化と名前空間の設計も重要です。適切なフォルダ構成やファイル命名規則を使用して、プロジェクト全体を整理しましょう。

大規模なアプリケーションを効果的に開発するためには、コードを小さな部品に分割し、それぞれの部品を組み合わせて機能を実現することが重要です。煩雑で理解しにくい1つのファイルに詰め込むことは避け、適切なモジュール化を行うことで、メンテナンス性と拡張性を確保しま

しょう。

絶対にやってはいけないこと5

無限ループを作成してプログラムをフリーズさせる。

無限ループを作成することは、プログラムを停止させることなく繰り返し処理を行うための一般的な手法です。しかし、適切に制御されない場合、無限ループはプログラムをフリーズさせる可能性があります。これにより、予期せぬ結果や深刻な問題が発生する可能性があります。

無限ループを作成してしまうと、プログラムはいつまでも同じ処理を繰

り返し続けるため、他の処理や入力の受付ができなくなります。これにより、他のプログラムやシステムの正常な動作にも悪影響を及ぼす可能性があります。また、無限ループが発生すると、プログラムを強制終了させることが困難になることもあります。

さらに、無限ループはCPUのリソースを無駄に使用するため、パフォーマンスの低下や動作の不安定化を引き起こす可能性があります。これは特に長時間の実行時や、複数の無限ループが同時に動作している場合に顕著です。

したがって、プログラミングにおいては無限ループを作成する際には、適切な制御や終了条件を設定することが重要です。必要なループ処理を適切に行いつつ、プログラムの正常な動作を保つためには、無限ループ

に陥らないように細心の注意を払う必要があります。

つまり、無限ループを作成してプログラムをフリーズさせることは、プログラムやシステムの動作を混乱させる可能性があり、避けるべき行為です。適切な制御と終了条件を設定し、正しくループ処理を行うことがポイントです。

絶対にやってはいけないこと6

例外を無視してエラーを握りつぶす。

Rubyプログラミングにおいて、例外を無視してエラーを握りつぶすことは絶対にやってはいけません。例外はプログラムの実行中に起こる予

期せぬ状況を表し、例外を適切に処理することはプログラムの安全性と信頼性の向上につながります。

例外を握りつぶすことは、問題の本質を見逃し、バグが混入する可能性を高めます。エラーを無視することで、本来なら修正すべき箇所を見過ごし、プログラム全体の品質が低下してしまうでしょう。

例外処理は、例外が発生した場合に適切な対処方法を実行するための仕組みです。例えば、例外をキャッチしてエラーメッセージを出力することで、ユーザーが原因を理解しやすくなります。また、例外を処理することで、プログラムの流れを制御し、予期せぬ動作を回避することも可能となります。

例外を握りつぶす代わりに、例外処理を適切に実装することを心がけましょう。例外発生時の対応策を考慮し、適切なメッセージを表示することで、プログラムの品質向上につながります。

絶対にやってはいけないこと7

適切なエラーメッセージを表示しない。

Rubyプログラムを書く上で、絶対にやってはいけないことの一つは、適切なエラーメッセージを表示しないことです。エラーメッセージは、プログラム内の問題やバグを特定し解決する際に非常に役立つものです。適切なエラーメッセージは、どのような問題が発生しているのか明確に示し、開発者に対して問題の解決をサポートします。

エラーメッセージが不十分だったり、意味不明なメッセージが表示されると、開発者は問題の原因を特定するのに困難を感じるかもしれません。このような場合、問題を解決するプロセスが長くなり、開発者の時間とエネルギーを浪費することになります。

したがって、Rubyプログラミングにおいては、適切なエラーメッセージを表示することは非常に重要です。エラーが発生した場合には、エラーメッセージに含まれる情報を活用し、問題を特定し解決していくようにしましょう。

例えば、以下のようなエラーメッセージは適切ではありません。

"エラーが発生しました。"

このメッセージだけでは、具体的な問題点が分からず、開発者が解決策を見つけるのは難しいでしょう。

一方で、以下のようなエラーメッセージは非常に役立ちます。

"undefined method `foo' for nil:NilClass"

このメッセージでは、存在しないメソッド"foo"がNilクラスに対して呼び出されていることが分かります。この情報をもとに、コードを修正することができます。

適切なエラーメッセージを表示することは、Rubyプログラミングにおいてメンテナビリティを高めるために不可欠なスキルです。問題の特定と解決をスムーズに行いたい場合には、十分な情報を含んだエラーメッセージを表示するように心がけましょう。

絶対にやってはいけないこと8

過度なコメントアウトを残す。

過度なコメントアウトを残すことは、Rubyプログラミングで絶対にやってはいけないことです。コメントアウトはプログラム内の特定のコードを無効化するために使用されますが、過剰なコメントアウトはコードの可読性を損ね、保守性を低下させる可能性があります。

多くのコメントアウトは、開発中に必要な情報やメモを記録するために使用されますが、これらのコメントは最終的なプロダクトのコードに含めるべきではありません。不必要なコメントアウトを残すことは、他の開発者がコードを理解するのを困難にする可能性があります。

コメントアウトの代わりに、適切なコードの変更やリファクタリングを行うことが推奨されます。コメントアウトされたコードが不要になった場合は、それを削除するか、適切に処理するようにしてください。

短期的な開発の便宜のためにコメントアウトを使用することはありますが、そのコードがプロダクトにデプロイされる際には、コメントアウトを削除することを忘れないようにしましょう。

以上から、過度なコメントアウトを残すことは避け、コードの可読性と保守性を高めるために、コメントアウトを適切に使用することが重要です。

絶対にやってはいけないこと9

他のプログラマーが理解できないような変数名を使用する。

他のプログラマーが理解できないような変数名を使用するのは絶対にやってはいけないことです。

変数名はプログラムの可読性とメンテナンス性を高めるために非常に重要です。他のプログラマーがコードを読んだとき、変数名からその意図や役割を理解することができるはずです。しかし、意味不明な変数名を使用すると、他の人がコードを理解するのに困難を強いるばかりでなく、自分自身でも後で見直す時に混乱することがあります。

たとえば、以下のような例を考えてみましょう。

```ruby
a = 10
b = 5
c = a + b
```

これでは、変数名がaやbといった一文字や意味不明な名前であるため、

このコードが何を行っているのか一目ではわかりません。一時的な変数

である場合でも、その意図がわかるような名前にすることが重要です。

```ruby
first_number = 10

second_number = 5

result = first_number + second_number
```

このように変数名を適切に付けることで、他のプログラマーや自分自身

がコードを理解しやすくなります。読みやすいコードはバグを見つける

のにも助けになり、プログラムの品質を向上させることができます。

変数名はできるだけ意味のある名前にし、その役割や目的が推測できるようにすることが重要です。他のプログラマーが理解できるような変数名を使用することで、コードのメンテナンス性を高め、より効率的で信頼性の高いプログラムを構築することができます。

絶対にやってはいけないこと10

長大な関数を作成する。

長大な関数を作成することは、Rubyプログラミングで絶対にやってはいけないことの一つです。長大な関数は、可読性を低下させ、保守性を

損ないます。また、デバッグが困難になり、ミスやバグを引き起こしやすくなります。

代わりに、関数を小さく、短く、シンプルに保つことを心掛けましょう。関数は特定のタスクや目的に集中するように設計されるべきです。関数が長大になってしまった場合は、それをさらに細かく分割することを検討しましょう。それによって、コードの理解や再利用が容易になります。

以下は、関数を分割する例です。

```ruby
# 長大な関数
```

```
def calculate_total
  # たくさんの処理...
end

# 分割した関数
def calculate_subtotal
  # サブトータルを計算する処理...
end

def calculate_tax
  # 税金を計算する処理...
end
```

```
def calculate_shipping_cost

  # 配送料を計算する処理...

end



def calculate_total

  subtotal = calculate_subtotal

  tax = calculate_tax

  shipping_cost = calculate_shipping_cost



  total = subtotal + tax + shipping_cost

  return total

end
```

このように関数を細かく分割することで、処理がより明確になり、コードの読みやすさと保守性が向上します。

絶対にやってはいけないこと11

コードの冗長性を高めるために同じコードを繰り返し使用する。

Rubyプログラミングで絶対にやってはいけないことは、コードの冗長性を高めるために同じコードを繰り返し使用することです。

コードの冗長性は、メンテナンス性や可読性を損なう可能性がありま

す。同じコードを繰り返し使用することで、修正や変更が必要な場合にすべての箇所を変更する必要が生じます。これにより、バグの原因となる可能性が高まります。

代わりに、同じコードが必要な場合は、メソッドや関数を作成し、コードを再利用するようにしてください。これにより、コードの一貫性が保たれ、修正や変更が必要な場合も一箇所で済みます。

また、Rubyでは、繰り返し処理を行うための便利なメソッドやイテレータが提供されています。これらの機能を活用することで、より効率的かつ簡潔なコードを実現することができます。

結論として、コードの冗長性を高めるために同じコードを繰り返し使用

することは避け、メソッドやイテレータを活用して効率的かつメンテナンス性の高いコードを書くことが重要です。

絶対にやってはいけないこと12

パスワードや秘密鍵などの機密情報をソースコードにハードコーディングする。

パスワードや秘密鍵などの機密情報をソースコードにハードコーディングすることは、Rubyプログラミングで絶対にやってはいけないことです。

機密情報をソースコードに直接書き込むことは、セキュリティ上のリス

クを引き起こします。ソースコードは通常、バージョン管理システムによって共有されるため、機密情報が漏洩する危険性が高まります。また、ソースコードが不要なまでに大量の機密情報を保持することは、メンテナンスやセキュリティ管理の観点からも非効率です。

代わりに、機密情報は環境変数や設定ファイルなど、外部の安全なストレージに保存することをお勧めします。このような方法を使用することで、機密情報をソースコードとは分離させ、管理しやすく、セキュリティレベルを高めることができます。

例えば、環境変数を使用する場合は、Rubyの`ENV`モジュールを利用して、環境変数から値を取得することができます。

```ruby

password = ENV['MY_PASSWORD']

```

設定ファイルを使用する場合は、専用の設定ファイルを作成し、そのファイルから機密情報を読み込むようにします。

```ruby

require 'yaml'

config = YAML.load_file('config.yaml')

password = config['password']

```

このような方法を使用することで、機密情報を安全に管理し、プログラムのセキュリティを保護することができます。機密情報をソースコードにハードコーディングすることは避け、最善のセキュリティプラクティスに則って開発を行いましょう。

絶対にやってはいけないこと13

無駄に複雑な正規表現を使用する。

無駄に複雑な正規表現を使用することは、Rubyプログラミングにおいて絶対にやってはいけないことです。正規表現は非常に強力で便利なツールですが、使い方を間違えるとコードの可読性を低下させたり、パフ

ォーマンスの問題を引き起こす可能性があります。

例えば、単純な文字列のマッチングを行う場合でも、複雑な正規表現パターンを使ってしまうと、その処理に時間がかかったり、メモリの使用量が増えたりする可能性があります。また、他のメンバーがあなたのコードを理解するのにも困難を生じさせることがあります。

また、正規表現のパターンが複雑になるほど、バグを含んだコードを書く可能性も高まります。特に、長大な正規表現では、どこが間違っているのかを特定することが難しく、デバッグに非常に時間がかかります。

したがって、正規表現を使用する場合は、処理を目的に合わせてシンプルに保つように心がけましょう。わかりやすく、効率的なコードを書く

ことで、将来的な保守性や可読性の向上にも繋がります。

絶対にやってはいけないこと14

ライブラリやgemを一切使用しない。

ライブラリやgemを一切使用しないことは、Rubyプログラミングにおいて絶対にやってはいけないことの一つです。なぜなら、Rubyの豊富なライブラリやgemは、プログラムをより簡単かつ効率的に開発するための非常に重要なツールです。

ライブラリやgemは、多くの場合、すでに実装された関数やメソッドを提供しています。これにより、自分で同じ機能を再実装する必要がなく

なります。また、ライブラリやgemは、パフォーマンスの最適化やセキュリティの向上など、高度な機能も提供しています。

一つ一つの機能を自分で実装しようとすると、時間がかかるだけでなく、バグの可能性も高まります。また、自分で機能を実装することで、再利用性が低下し、保守性も悪くなります。

さらに、ライブラリやgemを使用しないことにより、プログラムの機能が制限され、開発の幅も狭まります。これは、長期的な開発においても大きな制約となります。

そのため、Rubyプログラミングにおいては、ライブラリやgemを活用することが非常に重要です。効率的かつ高品質なコードを実現するため

には、既存のリソースを活用し、他の開発者が提供する優れたツールを積極的に利用するべきです。

結論として、Rubyプログラミングでは、ライブラリやgemを一切使用しないことは避けるべきです。これらのツールは、開発効率の向上や品質の向上に不可欠な要素であり、積極的に活用することが重要です。

絶対にやってはいけないこと15

グローバル変数を乱用する。

グローバル変数は、プログラムのどこからでもアクセスできる便利な機能ですが、誤った使い方をすると混乱を招きます。以下に、グローバル

変数を乱用することが引き起こす問題をいくつか説明します。

１．名前の衝突と競合：グローバル変数はプログラム全体で共有されるため、異なる箇所で同じ名前の変数を使用すると、予期せぬ競合が発生します。これにより、値が上書きされたり、プログラムの動作が不安定になったりする可能性があります。

２．変更の追跡の困難さ：グローバル変数の使用は、変数の値がどこで変更されているのかを追跡することを困難にします。特に大規模なプログラムでは、変数の値が予期せず変更されている場合に、バグの原因を特定するのが難しくなります。

３．テストの困難さ：グローバル変数を使用すると、関数やメソッドの

テストが困難になります。グローバル変数の値を変更することなく関数の結果をテストするのは難しいため、プログラム全体の正しさを確認することが難しくなります。

以上の理由から、グローバル変数を乱用することは避けるべきです。代わりに、できる限りローカル変数やクラス変数を使用し、変数のスコープを制限することをおすすめします。これにより、プログラムの信頼性を高め、保守性を向上させることができます。

絶対にやってはいけないこと16

マジックナンバーをコード内に散在させる。

マジックナンバーをコード内に散在させるのは、絶対にやってはいけないことです。マジックナンバーとは、プログラム内で具体的な値を表すために直接書かれた数値のことを指します。

マジックナンバーは、コードの理解やメンテナンスを困難にします。なぜなら、その数値の意味や目的が明確になっていないため、後からコードを見直す際に混乱を招く可能性があるからです。

また、マジックナンバーはプログラムの柔軟性にも悪影響を及ぼします。もし、その数値を変更する必要が生じた場合、全てのコード内のマジックナンバーを個別に探し出して修正しなければなりません。この作業は手間がかかり、ヒューマンエラーのリスクが高まります。

代わりに、マジックナンバーを意味のある定数や変数で置き換えるべきです。これにより、コードの可読性が向上し、後で修正が容易になります。また、定数や変数には適切な名前を付け、その数値がどのような役割を果たしているのかを明示させることで、コードの理解も容易になります。

したがって、マジックナンバーをコード内に散在させるのは避けるべきです。常に意味のある定数や変数を使用し、プログラムの可読性と保守性を高めましょう。

絶対にやってはいけないこと17

過度なジャンプ (goto) ステートメントを使用する。

Rubyにはジャンプステートメントであるgotoが存在しません。これは意図的な設計決定であり、プログラムの可読性と保守性を高めるためです。

しかし、あえてジャンプステートメントを模倣するために無理に方法を探すことは推奨されません。gotoステートメントはプログラムのフローを不必要に複雑化し、バグの温床となります。

代わりに、条件分岐やループ構造を適切に使用することで、プログラムの制御フローを明確化しましょう。これにより、コードの可読性を向上させ、将来のメンテナンスや拡張性への影響を最小限に抑えることができます。

また、Rubyでは例外処理を活用することで、プログラムの中断やエラー処理を効果的に行うことができます。これにより、プログラムの安定性と保守性が向上します。

過度なジャンプステートメントの使用は、Rubyプログラミングにおいては避けるべきです。正攻法でプログラミングし、シンプルで読みやすいコードを書くことを心掛けましょう。

絶対にやってはいけないこと18

ローカル変数の名前をキーワードと競合させる。

Rubyのキーワードは、言語の組み込み機能を表す特殊な単語です。これらのキーワードには予め定義された意味がありますので、変数やメソッドの名前として使用するべきではありません。

例えば、以下のようなコードがあったとします。

```ruby
class = "User"
```

このコードはコンパイルエラーとなります。なぜなら、`class`はRubyのキーワードであり、文法的に正しくないからです。

キーワードと変数名が競合すると、予期せぬ動作やエラーが発生する可能性があります。コードが読みにくくなるだけでなく、デバッグも困難になります。

そこで、ローカル変数を定義する際には、キーワードとは異なる名前を選ぶようにしましょう。変数名は意味のある名前にすることで、可読性の向上とバグの予防につながります。

正しく命名された変数は、プログラムの保守性を高めると同時に、他の

プログラマーが理解しやすいコードを書くことができます。キーワードとの競合を避けることで、よりスムーズな開発が可能になるでしょう。

絶対にやってはいけないこと19

非効率的なデータ構造を選択する。

プログラミングにおいて、非効率的なデータ構造を選択することは絶対にやってはいけません。データ構造は、プログラムの動作やパフォーマンスに直接影響を与えるため、適切なデータ構造の選択は非常に重要です。

例えば、配列を使用してデータを管理する場合、要素の追加や削除が頻

繁に行われる場合にはリストや連結リストを使用する方が効率的です。逆に、データの検索やソートが頻繁に行われる場合はバイナリツリーやハッシュテーブルを使用することで、処理速度を向上させることができます。

適切なデータ構造を選択しない場合、プログラムは無駄な処理を行うため、実行時間やメモリ使用量が増えてしまいます。これにより、プログラムのパフォーマンスが低下し、ユーザー体験やシステムの安定性に悪影響を及ぼす可能性があります。

そのため、データ構造を選択する際には、データの特性や操作の頻度などを考慮し、効率的なデータ構造を選ぶようにしましょう。また、データの追加や削除、検索、ソートなどの操作について、各データ構造の特

性やアルゴリズムについて学ぶことも重要です。

非効率的なデータ構造を選択することは、プログラムのパフォーマンスに直結するため、絶対に避けるべきです。しっかりとしたデータ構造の選択は、より高速で効率的なプログラムの開発につながります。

絶対にやってはいけないこと20

プロジェクト内で一貫性のないコーディングスタイルを採用する。

プロジェクト内で一貫性のないコーディングスタイルを採用することは、Rubyプログラムの品質と保守性に深刻な問題を引き起こす可能性

があります。以下に、一貫性のないコーディングスタイルがもたらす潜在的な問題点を紹介します。

1. メンテナンスの困難さ: 異なるスタイルで書かれたコードは、他の開発者が理解しにくくなります。コードベースが複数のコーディングスタイルで溢れていると、必要な変更を行うために時間を浪費することになるでしょう。

2. バグの発生リスク: 一貫性のないコーディングスタイルは、不必要なバグやエラーの可能性を高めます。たとえば、変数の名前が一貫していない場合、間違った変数を参照する可能性があります。コーディングスタイルの一貫性は、バグの発生リスクを最小限に抑えるために重要です。

3. チームワークの阻害: プロジェクトでは複数の開発者が協力して作業を進めることが一般的です。しかし、一貫性のないコーディングスタイルでは、チーム内での協力が困難になります。コードがばらばらのスタイルで書かれると、他の開発者が自分のコードに適合させる必要が生じます。

以上の理由から、プロジェクト内で一貫性のないコーディングスタイルを採用することは避けましょう。チーム全体でルールを共有し、統一されたコーディングスタイルを遵守することで、コードの品質と保守性を向上させることができます。

絶対にやってはいけないこと21

コメントを一切書かない。

コメントを一切書かないことは、Rubyプログラミングにおいて絶対にやってはいけないことです。コメントは、プログラムの読みやすさやメンテナンス性を向上させるために非常に重要な役割を果たします。

コメントを書かずにプログラムを作成すると、他の開発者がコードを理解するのが困難になります。特に大規模なプロジェクトでは、複数の開発者が関わることが多く、他の人が書いたコードを理解する必要が生じます。コメントがないと、プログラムの意図や動作が不明瞭になり、バグの原因を特定するのも難しくなります。

また、コメントは自分自身が後でコードを読み直す際にも役立ちます。

コードの意図や目的を忘れることがありますが、コメントを活用するこ

とで、過去の自分が書いたコードを理解するのに役立ちます。

コメントを書くことは、プログラミングの基本的な実践です。プログラ

ムを読みやすく、メンテナンス性を高めるために、コメントを正しく活

用してください。それによって、他の開発者や自分自身がプログラムを

理解しやすくなります。

絶対にやってはいけないこと22

コード内にアートワークやエモジを埋め込む。

Rubyプログラミングで絶対にやってはいけないことの一つは、コード内にアートワークやエモジを埋め込むことです。プログラムは効率的かつ正確に動作するために設計されており、冗長な要素や非関連のコンテンツは避けるべきです。コード内にアートワークやエモジを使用すると、可読性が低下し、他の開発者がコードを理解するのが難しくなります。また、一部のエディタや開発環境ではアートワークやエモジを正しく表示できない場合もあります。そのため、Rubyプログラミングではコード内にアートワークやエモジを埋め込むことは避けるべきです。代わりに、コードの可読性を高めるために意味のある変数名やコメントを使用することをおすすめします。

絶対にやってはいけないこと23

セキュリティホールを意図的に作成する。

Rubyプログラミングで絶対にやってはいけないことは、セキュリティホールを意図的に作成することです。

セキュリティホールは、システムの脆弱性を攻撃者に利用される可能性がある隙間や弱点です。セキュリティホールを意図的に作成することは、システムやデータへの不正アクセスや悪意のある攻撃のリスクを高める行為です。

Rubyプログラミングでは、セキュリティを最優先に考える必要があり

ます。そのため、セキュリティホールを意図的に作成することは厳禁です。代表的な例として、以下のような行為があります。

1. インプットの検証やバリデーションを行わない

2. ユーザー入力をそのままSQLクエリやシェルコマンドに組み込む

3. セキュリティ設定やアクセス制御を適切に行わない

これらの行為は、悪意のあるユーザーによるセキュリティ侵害やデータ漏洩の可能性を高めます。セキュリティホールを意図的に作成することは、開発者としての責任を果たさない行為であり、法的な問題にもなりかねません。

セキュリティは常に進化している領域であり、特にWebアプリケーショ

ンの場合は、定期的に脆弱性テストやセキュリティアップデートを行うことが重要です。セキュリティ意識を持ち、信頼性の高いシステムを作り上げるために、セキュリティホールを意図的に作成することは絶対に避けるべきです。

絶対にやってはいけないこと24

意味のない無限ループを作成してCPUを占有する。

絶対にやってはいけないことの一つに、意味のない無限ループを作成してCPUを占有することがあります。これは、システムの正常な動作を阻害し、重大な問題を引き起こす可能性があります。

無限ループとは、プログラムが指定された条件を満たす限り繰り返し実行される処理のことです。通常、適切な条件を設定してループを作成することはプログラミングの一環であり、効果的なコードの作成に不可欠です。しかし、条件を満たすことなく無限にループが続く場合、プログラムは終了せず、CPUリソースを浪費し続けます。

CPUはコンピュータの中核であり、各種の処理を実行するために使用されます。しかし、CPUリソースは有限であるため、長時間にわたってCPUを占有することは、他のプログラムやプロセスの正常な実行を妨げる可能性があります。特に、サーバーや共有システムの場合は、他のユーザーにも影響を及ぼすことが考えられます。

例えば、以下のような意味のない無限ループのコードを作成してしまうと、予期せぬ結果をもたらす可能性があります。

```ruby
while true
  # 何もしない
end
```

この無限ループは、条件が常に `true` であるため、プログラムは永遠に実行され続けます。このようなコードを実行すると、CPUリソースが完全に占有され、他のプロセスやタスクが停止してしまうかもしれません。

したがって、プログラミングにおいては、意図的に無限ループを作成してCPUを占有することは避けるべきです。適切な条件設定や処理の最適化を行い、プログラムの効率性と正常な動作を確保するようにしましょう。

絶対にやってはいけないこと25

ソースコードを暗号化して共有し、他のプログラマーが協力できないようにする。

ソースコードを暗号化することは、Rubyプログラミングにおいて絶対にやってはいけないことです。プログラマーは協力して仕事をするため

には、コードを理解して変更できる必要があります。しかし、ソースコードを暗号化してしまうと、他のプログラマーがコードを理解できなくなってしまいます。

プログラミングはコミュニケーションの一環であり、他の人と協力して問題を解決することが重要です。ソースコードを暗号化してしまうと、他のプログラマーがコードを読んだり修正したりすることができなくなります。これは、プロジェクトの進行を滞らせ、問題解決のスピードを遅らせることになります。

また、ソースコードを暗号化することはセキュリティ上のリスクともなります。誰もがアクセスできないようにするつもりでも、暗号解読の方法を知る人物によってコードが解読される可能性があるからです。暗号

化することは逆にセキュリティを脆弱にする可能性があります。

ソースコードを他のプログラマーと共有する際には、分かりやすく、メンテナンスしやすいようにすることが重要です。変数名や関数名を適切に設定し、コメントやドキュメンテーションを充実させることで、他のプログラマーがコードを理解しやすくなります。

総括すると、ソースコードを暗号化して共有することは、Rubyプログラミングにおいて絶対にやってはいけないことです。プログラマー間のコミュニケーションと協力が重要であり、暗号化はその邪魔をするだけです。コードの可読性とメンテナンス性を高めることが大切ですので、暗号化は避けましょう。

Rubyプログラミングで絶対にやってはいけない25のこと

著者　尺一麟

発行日　令和5年9月9日