Signal & Slot リファレンス

PySide6におけるシグナル・スロット機能についての詳細なリファレンス資料です。

概要

シグナル・スロットは、Qtのコア機能の一つで、オブジェクト間の安全で柔軟な通信を提供します。

基本概念

シグナル (Signal)

- イベント発生時に送出される
- 引数を持つことができる
- 複数のスロットに接続可能

スロット (Slot)

- シグナルを受信する関数・メソッド
- 通常のPython関数として定義可能
- 戻り値は無視される

基本的な使用方法

シグナルの定義

```python
from PySide6.QtCore import QObject, Signal

class MyClass(QObject):
    # 引数なしシグナル
    finished = Signal()

    # 引数ありシグナル
    dataChanged = Signal(str)
    valueUpdated = Signal(int, str)

    # オーバーロードシグナル
    progress = Signal(int)
    progress = Signal(str)
```

シグナルとスロットの接続

```python
from PySide6.QtWidgets import QPushButton

# 基本的な接続
button = QPushButton("Click me")
button.clicked.connect(my_function)

# メソッドとの接続
button.clicked.connect(self.handle_click)

# ラムダ関数との接続
button.clicked.connect(lambda: print("Button clicked!"))

# 引数付きシグナルの接続
line_edit.textChanged.connect(self.on_text_changed)
```

接続の解除

```python
# 特定の接続を解除
button.clicked.disconnect(my_function)

# すべての接続を解除
button.clicked.disconnect()

# オブジェクトからのすべての接続を解除
button.disconnect()
```

実用的な例

カスタムシグナルの使用

```
 from PySide6.QtCore import QObject, Signal
from PySide6.QtWidgets import QWidget, QPushButton, QVBoxLayout

class Counter(QObject):
    """カウンター値の変更を通知するクラス"""

    # シグナルの定義
    valueChanged = Signal(int)
    limitReached = Signal()

    def __init__(self):
        super().__init__()
        self._value = 0
        self._limit = 10

    def increment(self):
        self._value += 1
        self.valueChanged.emit(self._value)  # シグナル送出

        if self._value >= self._limit:
            self.limitReached.emit()

    def reset(self):
        self._value = 0
        self.valueChanged.emit(self._value)

class CounterWidget(QWidget):
    def __init__(self):
        super().__init__()
        self.setup_ui()
        self.setup_connections()

    def setup_ui(self):
        layout = QVBoxLayout()
        self.increment_btn = QPushButton("増加")
        self.reset_btn = QPushButton("リセット")
        layout.addWidget(self.increment_btn)
        layout.addWidget(self.reset_btn)
        self.setLayout(layout)

        self.counter = Counter()

    def setup_connections(self):
        # ボタンのクリックイベントとカウンターを接続
        self.increment_btn.clicked.connect(self.counter.increment)
        self.reset_btn.clicked.connect(self.counter.reset)

        # カウンターの変更イベントとUIの更新を接続
        self.counter.valueChanged.connect(self.update_display)
        self.counter.limitReached.connect(self.handle_limit_reached)

    def update_display(self, value):
        self.setWindowTitle(f"カウンター: {value}")

    def handle_limit_reached(self):
        print("制限値に達しました！")
```

データバインディングの実装

```
 from PySide6.QtCore import QObject, Signal, Property

class Model(QObject):
    """データモデルクラス"""

    nameChanged = Signal(str)
    ageChanged = Signal(int)

    def __init__(self):
        super().__init__()
        self._name = ""
        self._age = 0

    @Property(str, notify=nameChanged)
    def name(self):
        return self._name

    @name.setter
    def name(self, value):
        if self._name != value:
            self._name = value
            self.nameChanged.emit(value)

    @Property(int, notify=ageChanged)
    def age(self):
        return self._age

    @age.setter
    def age(self, value):
        if self._age != value:
```

```
            self._age = value
            self.ageChanged.emit(value)

class View(QWidget):
    """ビュークラス"""

    def __init__(self, model):
        super().__init__()
        self.model = model
        self.setup_ui()
        self.setup_bindings()

    def setup_bindings(self):
        # モデルの変更をビューに反映
        self.model.nameChanged.connect(self.name_edit.setText)
        self.model.ageChanged.connect(self.age_spinbox.setValue)

        # ビューの変更をモデルに反映
        self.name_edit.textChanged.connect(self.model.name.fset)
        self.age_spinbox.valueChanged.connect(self.model.age.fset)
```

高度な機能

シグナルのブロック

```
# シグナルを一時的にブロック
button.blockSignals(True)
button.setText("新しいテキスト")  # textChangedシグナルが発生しない
button.blockSignals(False)

# コンテキストマネージャーとして使用
from contextlib import contextmanager

@contextmanager
def blocked_signals(obj):
    old_state = obj.blockSignals(True)
    try:
        yield obj
    finally:
        obj.blockSignals(old_state)

# 使用例
with blocked_signals(button):
    button.setText("一時的なテキスト")
```

カスタム接続タイプ

```
from PySide6.QtCore import Qt

# 異なる接続タイプ
button.clicked.connect(slot, Qt.ConnectionType.DirectConnection)
button.clicked.connect(slot, Qt.ConnectionType.QueuedConnection)
button.clicked.connect(slot, Qt.ConnectionType.AutoConnection)
```

シグナルチェーン

```
class SignalChain(QObject):
    step1Completed = Signal()
    step2Completed = Signal()
    allCompleted = Signal()

    def __init__(self):
        super().__init__()
        # シグナルをチェーン接続
        self.step1Completed.connect(self.start_step2)
        self.step2Completed.connect(self.all_completed)

    def start_process(self):
        # 処理開始
        self.step1Completed.emit()

    def start_step2(self):
        # ステップ2実行
        self.step2Completed.emit()

    def all_completed(self):
        self.allCompleted.emit()
```

デバッグとトラブルシューティング

接続状態の確認

```
from PySide6.QtCore import QObject

# 接続数の確認
connections = QObject.receivers(button.clicked)
```

```
print(f"接続数: {connections}")

# デバッグ情報の出力
import os
os.environ['QT_LOGGING_RULES'] = 'qt.qpa.debug=true'
```

よくある問題と解決策

```
# 1. オブジェクトの生存期間の問題
class BadExample:
    def __init__(self):
        button = QPushButton()
        button.clicked.connect(self.handle_click)
        # buttonがスコープを出ると削除される可能性

class GoodExample:
    def __init__(self):
        self.button = QPushButton()  # インスタンス変数として保持
        self.button.clicked.connect(self.handle_click)

# 2. 循環参照の回避
class Parent(QObject):
    def __init__(self):
        super().__init__()
        self.child = Child(self)

class Child(QObject):
    def __init__(self, parent):
        super().__init__(parent)  # 親を設定
        # 親への参照は弱参照として扱われる
```

パフォーマンス最適化

効率的なシグナル使用

```
# 頻繁に発生するシグナルの最適化
from PySide6.QtCore import QTimer

class OptimizedSignal(QObject):
    dataChanged = Signal()

    def __init__(self):
        super().__init__()
        self._pending_update = False
        self._timer = QTimer()
        self._timer.timeout.connect(self._emit_delayed)
        self._timer.setSingleShot(True)

    def request_update(self):
        if not self._pending_update:
            self._pending_update = True
            self._timer.start(100)  # 100ms後に実際のシグナルを送出

    def _emit_delayed(self):
        self._pending_update = False
        self.dataChanged.emit()
```

ベストプラクティス

1. **命名規則**: シグナル名は動詞形（clicked, finished等）を使用
2. **引数の設計**: 必要最小限の情報のみ渡す
3. **接続の管理**: 適切なタイミングでdisconnectを呼ぶ
4. **エラーハンドリング**: スロット内でのエラーハンドリングを適切に行う

```
class BestPracticeExample(QObject):
    # 適切なシグナル名
    fileLoaded = Signal(str)  # ファイルパスを渡す
    progressChanged = Signal(int)  # 進行率（0-100）

    def __init__(self):
        super().__init__()

    def safe_slot(self, data):
        """安全なスロットの実装例"""
        try:
            # 処理内容
            self.process_data(data)
        except Exception as e:
            print(f"エラーが発生しました: {e}")
            # エラー用シグナルを送出することも可能
```

参考リンク

- Qt Documentation - Signals & Slots

- PySide6 Signals & Slots