PytoWeb Framework Source Code

This document contains the source code of the PytoWeb Framework.

__init__.py

```
"""
PytoWeb - Create web interfaces using pure Python

This module provides a comprehensive framework for building web
interfaces using pure Python.
Version: 0.1.0
Author: PytoWeb Team
License: MIT
"""

from .app import App
from .router import Router
from .server import Server
from .components import Component, Button, Container, Text, Link,
Image
from .layouts import Grid, Flex
from .themes import Theme
from .animations import (
    Animation,
    AnimationManager,
    FADE_IN,
    FADE_OUT,
    SLIDE_IN,
    SLIDE_OUT,
    SLIDE_UP,
    SLIDE_DOWN,
    ROTATE,
    SCALE,
    BOUNCE,
    SHAKE,
    PULSE,
    ELASTIC_IN,
```

```
    ELASTIC_OUT,
    SWING,
    WOBBLE,
    ZOOM_IN,
    ZOOM_OUT
)
from .events import EventBridge, EventDelegate
from .vdom import VDOMRenderer

__version__ = "0.1.0"
__author__ = "PytoWeb Team"
__license__ = "MIT"

__all__ = [
    # Core
    'App',
    'Router',
    'Server',
    'VDOMRenderer',

    # Components
    'Component',
    'Button',
    'Container',
    'Text',
    'Link',
    'Image',

    # Layouts
    'Grid',
    'Flex',

    # Themes
    'Theme',

    # Animations
    'Animation',
    'AnimationManager',
    'FADE_IN',
    'FADE_OUT',
```

```
    'SLIDE_IN',
    'SLIDE_OUT',
    'SLIDE_UP',
    'SLIDE_DOWN',
    'ROTATE',
    'SCALE',
    'BOUNCE',
    'SHAKE',
    'PULSE',
    'ELASTIC_IN',
    'ELASTIC_OUT',
    'SWING',
    'WOBBLE',
    'ZOOM_IN',
    'ZOOM_OUT',

    # Events
    'EventBridge',
    'EventDelegate'
]
```

animations.py

```
"""
Animation system for PytoWeb
"""

from typing import Dict, Any, List, Optional, Union, Tuple
from dataclasses import dataclass
from .styles import Style

@dataclass
class AnimationTiming:
    """Animation timing configuration"""
    duration: float = 0.3
    delay: float = 0
    iteration_count: Union[int, str] = 1
    direction: str = 'normal'
```

```
    timing_function: str = 'ease'
    fill_mode: str = 'forwards'

class Animation:
    """Animation definition class"""

    def __init__(self, name: str, keyframes: Dict[str, Dict[str,
str]], timing: Optional[AnimationTiming] = None):
        self.name = name
        self.keyframes = keyframes
        self.timing = timing or AnimationTiming()

    def to_css(self) -> str:
        """Convert animation to CSS"""
        css = [f'@keyframes {self.name} {{']

        for selector, styles in self.keyframes.items():
            css.append(f'  {selector} {{')
            for prop, value in styles.items():
                css.append(f'    {prop}: {value};')
            css.append('  }')

        css.append('}')
        return '\n'.join(css)

    def get_animation_css(self) -> str:
        """Get animation CSS properties"""
        return (
            f'animation: {self.name} '
            f'{self.timing.duration}s '
            f'{self.timing.timing_function} '
            f'{self.timing.delay}s '
            f'{self.timing.iteration_count} '
            f'{self.timing.direction} '
            f'{self.timing.fill_mode}'
        )

class AnimationSequence:
    """Animation sequence for chaining multiple animations"""
    def __init__(self, *animations: Tuple[Animation, float]):
```

```
        self.animations = animations
        self.total_duration = sum(duration for _, duration in
animations)

    def to_css(self) -> str:
        """Convert animation sequence to CSS"""
        css_parts = []
        current_time = 0

        for animation, duration in self.animations:
            start_percent = (current_time / self.total_duration)
* 100
            end_percent = ((current_time + duration) /
self.total_duration) * 100

            for selector, styles in animation.keyframes.items():
                if selector == 'from':
                    selector = f'{start_percent}%'
                elif selector == 'to':
                    selector = f'{end_percent}%'
                else:
                    # 调整中间关键帧的时间点
                    original_percent = float(selector.replace('%',
''))
                    adjusted_percent = start_percent +
(original_percent / 100) * (end_percent - start_percent)
                    selector = f'{adjusted_percent}%'

                css_parts.append(f'  {selector} {{')
                for prop, value in styles.items():
                    css_parts.append(f'    {prop}: {value};')
                css_parts.append('  }')

            current_time += duration

        return '@keyframes ' + self.name + ' {\n' +
'\n'.join(css_parts) + '\n}'

class Flip(Animation):
```

```
    """3D flip animation"""
    def __init__(self, direction: str = 'x', duration: float =
0.6):
        timing = AnimationTiming(duration=duration)
        axis = 'X' if direction.lower() == 'x' else 'Y'

        super().__init__(f'flip-{direction}', {
            'from': {
                'transform': f'perspective(400px)
rotate{axis}(0)',
                'animation-timing-function': 'ease-out'
            },
            '40%': {
                'transform': f'perspective(400px)
translate{axis}(0) rotate{axis}(-20deg)',
                'animation-timing-function': 'ease-out'
            },
            '60%': {
                'transform': f'perspective(400px)
rotate{axis}(10deg)',
                'animation-timing-function': 'ease-in'
            },
            '80%': {
                'transform': f'perspective(400px) rotate{axis}(-
5deg)',
                'animation-timing-function': 'ease-in'
            },
            'to': {
                'transform': f'perspective(400px)',
                'animation-timing-function': 'ease-in'
            }
        }, timing)

class Elastic(Animation):
    """Elastic animation"""
    def __init__(self, direction: str = 'in', duration: float =
1.0):
        timing = AnimationTiming(duration=duration)

        if direction == 'in':
```

```
            keyframes = {
                '0%': {'transform': 'scale(0)'},
                '55%': {'transform': 'scale(1.05)'},
                '75%': {'transform': 'scale(0.95)'},
                '90%': {'transform': 'scale(1.02)'},
                '100%': {'transform': 'scale(1)'}
            }
        else:  # out
            keyframes = {
                '0%': {'transform': 'scale(1)'},
                '25%': {'transform': 'scale(0.95)'},
                '50%': {'transform': 'scale(1.05)'},
                '75%': {'transform': 'scale(0.95)'},
                '100%': {'transform': 'scale(0)'}
            }

        super().__init__(f'elastic-{direction}', keyframes,
timing)

class Swing(Animation):
    """Swing animation"""
    def __init__(self, duration: float = 1.0):
        timing = AnimationTiming(duration=duration)
        super().__init__('swing', {
            '0%': {'transform': 'rotate(0deg)'},
            '20%': {'transform': 'rotate(15deg)'},
            '40%': {'transform': 'rotate(-10deg)'},
            '60%': {'transform': 'rotate(5deg)'},
            '80%': {'transform': 'rotate(-5deg)'},
            '100%': {'transform': 'rotate(0deg)'}
        }, timing)

class Wobble(Animation):
    """Wobble animation"""
    def __init__(self, duration: float = 1.0):
        timing = AnimationTiming(duration=duration)
        super().__init__('wobble', {
            '0%': {'transform': 'translateX(0%)'},
            '15%': {'transform': 'translateX(-25%) rotate(-
5deg)'},
```

```
            '30%': {'transform': 'translateX(20%) rotate(3deg)'},
            '45%': {'transform': 'translateX(-15%) rotate(-
3deg)'},
            '60%': {'transform': 'translateX(10%) rotate(2deg)'},
            '75%': {'transform': 'translateX(-5%) rotate(-1deg)'},
            '100%': {'transform': 'translateX(0%)'}
        }, timing)

class TypeWriter(Animation):
    """Typewriter text animation"""
    def __init__(self, text_length: int, duration: float = 2.0):
        timing = AnimationTiming(duration=duration)
        steps = {}

        for i in range(text_length + 1):
            percentage = (i / text_length) * 100
            steps[f'{percentage}%'] = {
                'width': f'{i}ch',
                'border-right-color': f'{"transparent" if i ==
text_length else "currentColor"}'
            }

        super().__init__('typewriter', steps, timing)

class FadeIn(Animation):
    """Fade in animation"""
    def __init__(self, duration: float = 0.3):
        timing = AnimationTiming(duration=duration)
        super().__init__('fade-in', {
            'from': {'opacity': '0'},
            'to': {'opacity': '1'}
        }, timing)

class FadeOut(Animation):
    """Fade out animation"""
    def __init__(self, duration: float = 0.3):
        timing = AnimationTiming(duration=duration)
        super().__init__('fade-out', {
            'from': {'opacity': '1'},
            'to': {'opacity': '0'}
```

```
        }, timing)

class Slide(Animation):
    """Slide animation"""
    def __init__(self, direction: str = 'left', duration: float =
0.3):
        timing = AnimationTiming(duration=duration)

        if direction == 'left':
            keyframes = {
                'from': {'transform': 'translateX(-100%)',
'opacity': '0'},
                'to': {'transform': 'translateX(0)', 'opacity':
'1'}
            }
        elif direction == 'right':
            keyframes = {
                'from': {'transform': 'translateX(100%)',
'opacity': '0'},
                'to': {'transform': 'translateX(0)', 'opacity':
'1'}
            }
        elif direction == 'up':
            keyframes = {
                'from': {'transform': 'translateY(-100%)',
'opacity': '0'},
                'to': {'transform': 'translateY(0)', 'opacity':
'1'}
            }
        else:  # down
            keyframes = {
                'from': {'transform': 'translateY(100%)',
'opacity': '0'},
                'to': {'transform': 'translateY(0)', 'opacity':
'1'}
            }

        super().__init__(f'slide-{direction}', keyframes, timing)

class Rotate(Animation):
```

```
    """Rotate animation"""
    def __init__(self, degrees: int = 360, duration: float = 0.3):
        timing = AnimationTiming(duration=duration)
        super().__init__('rotate', {
            'from': {'transform': 'rotate(0deg)'},
            'to': {'transform': f'rotate({degrees}deg)'}
        }, timing)

class Scale(Animation):
    """Scale animation"""
    def __init__(self, from_scale: float = 0, to_scale: float = 1,
duration: float = 0.3):
        timing = AnimationTiming(duration=duration)
        super().__init__('scale', {
            'from': {'transform': f'scale({from_scale})'},
            'to': {'transform': f'scale({to_scale})'}
        }, timing)

class Bounce(Animation):
    """Bounce animation"""
    def __init__(self, duration: float = 1.0):
        timing = AnimationTiming(duration=duration)
        super().__init__('bounce', {
            '0%': {'transform': 'translateY(0)'},
            '20%': {'transform': 'translateY(0)'},
            '40%': {'transform': 'translateY(-30px)'},
            '50%': {'transform': 'translateY(0)'},
            '60%': {'transform': 'translateY(-15px)'},
            '80%': {'transform': 'translateY(0)'},
            '100%': {'transform': 'translateY(0)'}
        }, timing)

class Shake(Animation):
    """Shake animation"""
    def __init__(self, intensity: int = 10, duration: float =
0.8):
        timing = AnimationTiming(duration=duration)
        keyframes = {}
        steps = 10
        for i in range(steps + 1):
```

```
            percentage = f"{(i * 100) // steps}%"
            if i % 2 == 0:
                keyframes[percentage] = {'transform':
f'translateX({intensity}px)'}
            else:
                keyframes[percentage] = {'transform':
f'translateX(-{intensity}px)'}
        keyframes['100%'] = {'transform': 'translateX(0)'}

        super().__init__('shake', keyframes, timing)

class Pulse(Animation):
    """Pulse animation"""
    def __init__(self, scale: float = 1.1, duration: float = 1.0):
        timing = AnimationTiming(duration=duration,
iteration_count='infinite')
        super().__init__('pulse', {
            '0%': {'transform': 'scale(1)'},
            '50%': {'transform': f'scale({scale})'},
            '100%': {'transform': 'scale(1)'}
        }, timing)

class AnimationManager:
    """Animation management class"""

    _animations: Dict[str, Animation] = {}

    @classmethod
    def register(cls, animation: Animation):
        """Register animation"""
        cls._animations[animation.name] = animation

    @classmethod
    def get(cls, name: str) -> Optional[Animation]:
        """Get registered animation"""
        return cls._animations.get(name)

    @classmethod
    def get_all_css(cls) -> str:
        """Get all animations CSS"""
```

```
        return '\n\n'.join(anim.to_css() for anim in
cls._animations.values())

    @classmethod
    def create_sequence(cls, *animations: Tuple[str, float]) ->
AnimationSequence:
        """Create animation sequence from registered
animations"""
        sequence = []
        for name, delay in animations:
            animation = cls.get(name)
            if animation:
                sequence.append((animation, delay))
        return AnimationSequence(*sequence)

# 预定义动画实例
FADE_IN = FadeIn()
FADE_OUT = FadeOut()
SLIDE_IN = Slide()
SLIDE_OUT = Slide('right')
SLIDE_UP = Slide('up')
SLIDE_DOWN = Slide('down')
ROTATE = Rotate()
SCALE = Scale()
BOUNCE = Bounce()
SHAKE = Shake()
PULSE = Pulse()
ELASTIC_IN = Elastic('in')
ELASTIC_OUT = Elastic('out')
SWING = Swing()
WOBBLE = Wobble()
ZOOM_IN = Scale(0, 1, 0.3)  # 从0缩放到1
ZOOM_OUT = Scale(1, 0, 0.3)  # 从1缩放到0

# 注册预定义动画
for animation in [
    FADE_IN, FADE_OUT,
    SLIDE_IN, SLIDE_OUT, SLIDE_UP, SLIDE_DOWN,
```

```
    ROTATE, SCALE, BOUNCE, SHAKE, PULSE,
    ELASTIC_IN, ELASTIC_OUT, SWING, WOBBLE,
    ZOOM_IN, ZOOM_OUT
]:
    AnimationManager.register(animation)
```

app.py

```
"""
PytoWeb 应用主类
"""

from __future__ import annotations
from typing import Optional, Any, Callable, List, Dict, Union,
TypeVar, TYPE_CHECKING
from dataclasses import dataclass
from .server import Server
from .router import Router
from .components import Component
from .vdom import VDOMRenderer
import logging
import sys
import traceback
from http import HTTPStatus

if TYPE_CHECKING:
    from .middleware import Middleware

T = TypeVar('T', bound='App')

class AppError(Exception):
    """PytoWeb 应用异常基类"""
    pass

@dataclass
class AppConfig:
    """应用配置"""
```

```
    host: str = "localhost"
    port: int = 8000
    debug: bool = False
    static_dir: str = "static"
    template_dir: str = "templates"
    secret_key: Optional[str] = None

class App:
    """PytoWeb 应用主类"""

    def __init__(self, config: Optional[AppConfig] = None):
        """初始化应用"""
        try:
            self.config = config or AppConfig()
            self.server = Server(self.config.host,
self.config.port)
            self.router = Router()
            self.root: Optional[Component] = None
            self.renderer = VDOMRenderer()
            self._logger = logging.getLogger(__name__)

            # 配置日志
            if self.config.debug:
                logging.basicConfig(
                    level=logging.DEBUG,
                    format='%(asctime)s - %(name)s -
 %(levelname)s - %(message)s'
                )

            # 设置静态文件目录
            self.server.static_dir = self.config.static_dir

            # 注册默认路由处理器
            self.router.add('/', self._handle_root)
            self.server.add_route('/', self.router.dispatch)

        except Exception as e:
            raise AppError(f"Failed to initialize application:
```

```
{e}") from e

    def _handle_root(self, request: Dict[str, Any]) -> str:
        """处理根路由请求"""
        print("[DEBUG] Handling root request")
        if self.root is None:
            raise AppError("No root component mounted")
        try:
            html = self.render(self.root)
            print(f"[DEBUG] Generated HTML length: {len(html)}")
            return html
        except Exception as e:
            print(f"[DEBUG] Error rendering root: {e}")
            raise AppError(f"Failed to render root: {e}") from e

    def mount(self: T, component: Component) -> T:
        """挂载根组件"""
        try:
            if not isinstance(component, Component):
                raise AppError("Component must be an instance of
Component")
            self.root = component
            return self
        except Exception as e:
            if isinstance(e, AppError):
                raise
            raise AppError(f"Failed to mount component: {e}")
from e

    def render(self, component: Component) -> str:
        """渲染组件"""
        try:
            if not isinstance(component, Component):
                raise AppError("Component must be an instance of
Component")

            vdom = component.render()
            html = self.renderer.render_to_string(vdom)
```

```
            return f"""
            <!DOCTYPE html>
            <html lang="en">
            <head>
                <meta charset="UTF-8">
                <meta name="viewport" content="width=device-width,
initial-scale=1.0">
                <title>PytoWeb App</title>
                <style>{self._get_styles()}</style>
                <script>{self._get_scripts()}</script>
            </head>
            <body>
                <div id="app">{html}</div>
            </body>
            </html>
            """
        except Exception as e:
            if isinstance(e, AppError):
                raise
            raise AppError(f"Failed to render component: {e}")
from e

    def _get_styles(self) -> str:
        """获取应用样式"""
        try:
            from .styles import get_global_styles
            return get_global_styles()
        except Exception as e:
            self._logger.error(f"Failed to get styles: {e}")
            return ""

    def _get_scripts(self) -> str:
        """获取应用脚本"""
        try:
            from .events import get_client_script
            return get_client_script()
        except Exception as e:
            self._logger.error(f"Failed to get scripts: {e}")
```

```
            return ""

    def run(self, host: str = "127.0.0.1", port: int = 8000,
debug: bool = False):
        """运行应用"""
        try:
            if debug:
                self._logger.setLevel(logging.DEBUG)
            self.server.run(host, port)
        except Exception as e:
            raise AppError(f"Failed to run application: {e}")
from e
```

components.py

```
"""
PytoWeb 组件系统

提供基础和高级 UI 组件，支持虚拟滚动、拖放等功能。
"""

from __future__ import annotations
from typing import (
    Dict, Any, Optional, Callable, List, Set,
    TypeVar, TypedDict, Union, TYPE_CHECKING
)
from collections import OrderedDict
import weakref
import logging
from .elements import Element
from .styles import Style
from .events import EventDelegate, Event
import time
import sys
import asyncio
import uuid
```

```
import traceback
from dataclasses import dataclass
from datetime import datetime
import json
from functools import wraps

if TYPE_CHECKING:
    from typing import Literal

# 配置日志
logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger(__name__)

# 类型别名
T = TypeVar('T')
OptionsType = List[Dict[str, str]]
EventHandler = Callable[..., None]
ComponentList = List['Component']
PropDict = Dict[str, Any]
StateDict = Dict[str, Any]

class ComponentCache:
    """组件缓存系统"""
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance

    def __init__(self):
        if not hasattr(self, 'initialized'):
            self._cache: OrderedDict[str, tuple[Any, float]] =
OrderedDict()
            self._max_size = 100  # 最大缓存项数
            self._max_memory = 100 * 1024 * 1024  # 最大内存使用
(100MB)
            self._ttl = 300  # 缓存过期时间(秒)
```

```
            self._current_memory = 0
            self._logger = logging.getLogger(__name__)
            self.initialized = True

    def get(self, key: str) -> Optional[Any]:
        """获取缓存的组件"""
        try:
            if key in self._cache:
                value, timestamp = self._cache[key]
                current_time = time.time()

                # 检查是否过期
                if current_time - timestamp > self._ttl:
                    self._cache.pop(key)
                    self._current_memory -= sys.getsizeof(value)
                    return None

                # 更新访问顺序和时间戳
                self._cache.move_to_end(key)
                self._cache[key] = (value, current_time)
                return value
        except Exception as e:
            self._logger.error(f"Error getting cached component:
{e}", exc_info=True)
        return None

    def set(self, key: str, value: Any):
        """缓存组件"""
        try:
            current_time = time.time()
            value_size = sys.getsizeof(value)

            # 检查单个值是否超过最大内存限制
            if value_size > self._max_memory:
                self._logger.warning(f"Value too large to cache:
{value_size} bytes")
                return
```

```
            # 如果已存在，先移除旧值
            if key in self._cache:
                old_value, _ = self._cache.pop(key)
                self._current_memory -= sys.getsizeof(old_value)

            # 清理过期和超出内存限制的缓存
            while self._cache and (
                len(self._cache) >= self._max_size or
                self._current_memory + value_size >
self._max_memory or
                current_time -
next(iter(self._cache.values()))[1] > self._ttl
            ):
                removed_key = next(iter(self._cache))
                removed_value, _ = self._cache.pop(removed_key)
                self._current_memory -=
sys.getsizeof(removed_value)

            # 添加新值
            self._cache[key] = (value, current_time)
            self._current_memory += value_size

        except Exception as e:
            self._logger.error(f"Error caching component: {e}",
exc_info=True)

    def clear(self):
        """清除缓存"""
        self._cache.clear()
        self._current_memory = 0

    def get_stats(self) -> dict:
        """获取缓存统计信息"""
        return {
            'size': len(self._cache),
            'memory_usage': self._current_memory,
```

```
            'max_size': self._max_size,
            'max_memory': self._max_memory,
            'ttl': self._ttl
        }

class Component:
    """所有组件的基类"""

    def __init__(self):
        self.props: PropDict = {}
        self.state: StateDict = {}
        self.children: ComponentList = []
        self.parent: Optional['Component'] = None
        self.style = Style()
        self.tag_name = "div"  # 默认标签
        self._cache = ComponentCache()
        self._logger = logging.getLogger(__name__)
        self._mounted = False
        self._destroyed = False

        # 生命周期事件
        self.on_before_mount = EventDelegate()
        self.on_mounted = EventDelegate()
        self.on_before_update = EventDelegate()
        self.on_updated = EventDelegate()
        self.on_before_destroy = EventDelegate()
        self.on_destroyed = EventDelegate()
        self.on_error = EventDelegate()

        # 状态变更事件
        self.on_state_change = EventDelegate()
        self.on_prop_change = EventDelegate()

        self._memo_cache = {}
        self._memo_deps = {}

        self._lazy_loaded = False
        self._lazy_loading = False
```

```
        self._lazy_error = None
        self._lazy_promise = None

    def set_prop(self, key: str, value: Any):
        """设置属性"""
        try:
            old_value = self.props.get(key)
            if old_value != value:
                self.props[key] = value
                self.on_prop_change(self, key, old_value, value)
                self._update()
        except Exception as e:
            self._logger.error(f"Error setting prop {key}: {e}",
exc_info=True)
            self.on_error(self, e)

    def set_state(self, key: str, value: Any):
        """设置状态"""
        try:
            old_value = self.state.get(key)
            if old_value != value:
                self.state[key] = value
                self.on_state_change(self, key, old_value, value)
                self._update()
        except Exception as e:
            self._logger.error(f"Error setting state {key}: {e}",
exc_info=True)
            self.on_error(self, e)

    def add_child(self, child: 'Component'):
        """添加子组件"""
        try:
            child.parent = self
            self.children.append(child)
            self._update()
        except Exception as e:
            self._logger.error(f"Error adding child: {e}",
exc_info=True)
```

```
            self.on_error(self, e)

    def remove_child(self, child: 'Component'):
        """移除子组件"""
        try:
            if child in self.children:
                child.parent = None
                self.children.remove(child)
                self._update()
        except Exception as e:
            self._logger.error(f"Error removing child: {e}",
exc_info=True)
            self.on_error(self, e)

    def mount(self):
        """组件挂载"""
        try:
            if not self._mounted:
                self.on_before_mount(self)
                self._mounted = True
                for child in self.children:
                    child.mount()
                self.on_mounted(self)
        except Exception as e:
            self._logger.error(f"Error mounting component: {e}",
exc_info=True)
            self.on_error(self, e)

    def unmount(self):
        """组件卸载"""
        try:
            if self._mounted and not self._destroyed:
                self.on_before_destroy(self)
                self._mounted = False
                self._destroyed = True
                for child in self.children:
                    child.unmount()
                self.on_destroyed(self)
```

```
        except Exception as e:
            self._logger.error(f"Error unmounting component: {e}",
exc_info=True)
            self.on_error(self, e)

    def _update(self):
        """更新组件"""
        try:
            if self._mounted and not self._destroyed:
                self.on_before_update(self)
                # 实际更新逻辑
                self.on_updated(self)
        except Exception as e:
            self._logger.error(f"Error updating component: {e}",
exc_info=True)
            self.on_error(self, e)

    def validate_props(self, prop_types: Dict[str, type]):
        """验证属性类型"""
        for key, expected_type in prop_types.items():
            if key in self.props:
                value = self.props[key]
                if not isinstance(value, expected_type):
                    raise TypeError(f"Prop '{key}' expected type
{expected_type.__name__}, got {type(value).__name__}")

    def validate_state(self, state_types: Dict[str, type]):
        """验证状态类型"""
        for key, expected_type in state_types.items():
            if key in self.state:
                value = self.state[key]
                if not isinstance(value, expected_type):
                    raise TypeError(f"State '{key}' expected type
{expected_type.__name__}, got {type(value).__name__}")

    def render(self):
        """渲染组件"""
```

```
        try:
            print(f"[DEBUG] Rendering component:
{self.__class__.__name__}")
            element = Element(self.tag_name)

            # 添加样式
            if self.style:
                element.style.update(self.style.get_all())
                print(f"[DEBUG] Added styles:
{self.style.get_all()}")

            # 添加子组件
            for child in self.children:
                try:
                    child_element = child.render()
                    if child_element:
                        element.add(child_element)
                        print(f"[DEBUG] Added child element:
{child.__class__.__name__}")
                    else:
                        print(f"[WARNING] Child
{child.__class__.__name__} rendered None")
                except Exception as e:
                    print(f"[ERROR] Failed to render child
{child.__class__.__name__}: {e}")
                    raise

            return element
        except Exception as e:
            print(f"[ERROR] Failed to render
{self.__class__.__name__}: {e}")
            raise

    def memo(self, key: str, fn: Callable[..., Any], *deps: Any)
-> Any:
        """记忆化计算结果

        Args:
```

```
            key: 缓存键名
            fn: 要记忆化的函数
            deps: 依赖项，当这些值变化时重新计算

        Returns:
            记忆化的计算结果
        """
        current_deps = tuple(deps)

        # 检查依赖是否变化
        if (key not in self._memo_cache or
            key not in self._memo_deps or
            self._memo_deps[key] != current_deps):

            # 重新计算并缓存结果
            self._memo_cache[key] = fn()
            self._memo_deps[key] = current_deps

        return self._memo_cache[key]

    def clear_memo(self, key: Optional[str] = None):
        """清除记忆化缓存

        Args:
            key: 要清除的特定缓存键,如果为 None 则清除所有缓存
        """
        if key is None:
            self._memo_cache.clear()
            self._memo_deps.clear()
        else:
            self._memo_cache.pop(key, None)
            self._memo_deps.pop(key, None)

    def lazy_load(self, loader: Callable[[], Awaitable[Any]]) ->
None:
        """懒加载组件内容
```

```
        Args:
            loader: 异步加载函数
        """
        if not self._lazy_loaded and not self._lazy_loading:
            self._lazy_loading = True
            self._lazy_promise =
asyncio.create_task(self._do_lazy_load(loader))

    async def _do_lazy_load(self, loader: Callable[[],
Awaitable[Any]]) -> None:
        """执行懒加载

        Args:
            loader: 异步加载函数
        """
        try:
            result = await loader()
            self._handle_lazy_load_success(result)
        except Exception as e:
            self._handle_lazy_load_error(e)

    def _handle_lazy_load_success(self, result: Any) -> None:
        """处理懒加载成功

        Args:
            result: 加载结果
        """
        self._lazy_loaded = True
        self._lazy_loading = False
        self._lazy_error = None
        self.state['lazy_result'] = result
        self._update()

    def _handle_lazy_load_error(self, error: Exception) -> None:
        """处理懒加载错误
```

```
        Args:
            error: 错误信息
        """
        self._lazy_loaded = False
        self._lazy_loading = False
        self._lazy_error = error
        self._update()

    def is_lazy_loaded(self) -> bool:
        """检查是否已完成懒加载"""
        return self._lazy_loaded

    def is_lazy_loading(self) -> bool:
        """检查是否正在懒加载"""
        return self._lazy_loading

    def get_lazy_error(self) -> Optional[Exception]:
        """获取懒加载错误信息"""
        return self._lazy_error

class AsyncComponentMixin:
    """为组件添加异步支持的 Mixin 类"""
    def __init__(self):
        super().__init__()
        self._cache = ComponentCache()
        self._pending_updates = {}

    async def update_async(self, **kwargs):
        """异步更新组件状态"""
        update_id = str(uuid.uuid4())
        self._pending_updates[update_id] = asyncio.Future()

        try:
            await self.on_before_update.emit_async()
            self.state.update(kwargs)
            await self.on_updated.emit_async()
            self._pending_updates[update_id].set_result(True)
```

```
            except Exception as e:
                self._pending_updates[update_id].set_exception(e)
            finally:
                del self._pending_updates[update_id]

    async def render_async(self):
        """异步渲染组件"""
        cache_key = self._get_cache_key()
        cached = self._cache.get(cache_key)
        if cached:
            return cached

        try:
            await self.on_before_mount.emit_async()
            result = await self._render_async_impl()
            await self.on_mounted.emit_async()

            self._cache.set(cache_key, result)
            return result
        except Exception as e:
            self.logger.error(f"Error in async rendering: {e}")
            raise

    async def _render_async_impl(self):
        """异步渲染实现"""
        raise NotImplementedError("Async components must
implement _render_async_impl")

class AsyncComponent(AsyncComponentMixin, Component):
    """异步组件基类"""
    pass

class Suspense(Component):
    """处理异步加载状态的组件"""
    def __init__(self,
                 component: AsyncComponent,
                 fallback: Optional[Component] = None,
                 error_fallback: Optional[Component] = None):
```

```
        super().__init__()
        self.set_prop('component', component)
        self.set_prop('fallback', fallback or
self._default_fallback())
        self.set_prop('error_fallback', error_fallback or
self._default_error())

        self.state.update({
            'loading': True,
            'error': None
        })

    def _default_fallback(self):
        """默认加载组件"""
        loading = Component()
        loading.tag_name = "div"
        loading.style.add(
            text_align="center",
            padding="1rem"
        )
        loading.set_text("Loading...")
        return loading

    def _default_error(self):
        """默认错误组件"""
        error = Component()
        error.tag_name = "div"
        error.style.add(
            color="red",
            text_align="center",
            padding="1rem"
        )
        error.set_text("An error occurred")
        return error

    async def render_async(self):
        """异步渲染"""
        try:
            if self.state['loading']:
```

```
                return self.props['fallback']

            result = await self.props['component'].render_async()
            self.state['loading'] = False
            return result
        except Exception as e:
            self.state['error'] = str(e)
            self.logger.error(f"Error in Suspense: {e}")
            return self.props['error_fallback']

class ErrorBoundary(Component):
    """错误边界组件，用于捕获和处理子组件中的错误"""

    def __init__(self,
                 children: list[Component],
                 fallback: Optional[Callable[[Exception],
Component]] = None):
        super().__init__()
        self.set_prop('children', children)
        self.set_prop('fallback', fallback or
self._default_fallback)

        self.state.update({
            'error': None,
            'error_info': None
        })

        self._error_handler = ErrorHandler.get_instance()

    def _default_fallback(self, error: Exception) -> Component:
        """默认错误回退组件"""
        error_component = Component()
        error_component.tag_name = "div"
        error_component.style.add(
            color="red",
            padding="1rem",
            border="1px solid red",
            margin="1rem",
            background_color="rgba(255,0,0,0.1)"
```

```
        )
        error_component.set_text(f"Error: {str(error)}")
        return error_component

    def render(self):
        """渲染错误边界"""
        if self.state['error']:
            error_component =
self.props['fallback'](self.state['error'])
            return error_component

        try:
            return self.props['children']
        except Exception as e:
            self.state['error'] = e
            self.state['error_info'] =
self._error_handler._get_error_context()
            self._error_handler.handle_error(e,
self.state['error_info'])
            return self.props['fallback'](e)

@dataclass
class ErrorContext:
    """错误上下文信息"""
    component: Optional[str] = None
    function: Optional[str] = None
    line_number: Optional[int] = None
    file_path: Optional[str] = None
    stack_trace: Optional[str] = None
    additional_info: Dict[str, Any] = None

@dataclass
class ErrorReport:
    """详细错误报告"""
    error_type: str
    message: str
    context: ErrorContext
    timestamp: datetime
```

```
    severity: str
    handled: bool

class ErrorHandler:
    """中央错误处理系统"""
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance

    def __init__(self):
        if not hasattr(self, 'initialized'):
            self.error_listeners: List[Callable[[ErrorReport],
None]] = []
            self.error_history: List[ErrorReport] = []
            self.max_history = 100
            self.logger = logging.getLogger('pytoweb.errors')
            self.initialized = True

    @classmethod
    def get_instance(cls):
        return cls()

    def add_listener(self, listener: Callable[[ErrorReport],
None]):
        """添加错误监听器"""
        self.error_listeners.append(listener)

    def remove_listener(self, listener: Callable[[ErrorReport],
None]):
        """移除错误监听器"""
        self.error_listeners.remove(listener)

    def handle_error(self, error: Exception, context:
Optional[ErrorContext] = None):
        """处理错误"""
```

```
        if context is None:
            context = self._get_error_context()

        report = ErrorReport(
            error_type=type(error).__name__,
            message=str(error),
            context=context,
            timestamp=datetime.now(),
            severity=self._get_error_severity(error),
            handled=True
        )

        self.error_history.append(report)
        if len(self.error_history) > self.max_history:
            self.error_history.pop(0)

        for listener in self.error_listeners:
            try:
                listener(report)
            except Exception as e:
                self.logger.error(f"Error in error listener: {e}")

        self.logger.error(f"Error: {report.message}",
exc_info=True)

    def _get_error_context(self) -> ErrorContext:
        """从当前异常获取上下文"""
        tb = sys.exc_info()[2]
        while tb.tb_next:
            tb = tb.tb_next

        frame = tb.tb_frame
        return ErrorContext(
            function=frame.f_code.co_name,
            line_number=tb.tb_lineno,
            file_path=frame.f_code.co_filename,
            stack_trace=traceback.format_exc()
        )
```

```
    def _get_error_severity(self, error: Exception) -> str:
        """确定错误严重性"""
        if isinstance(error, (SystemError, MemoryError)):
            return "CRITICAL"
        if isinstance(error, (ValueError, TypeError)):
            return "ERROR"
        return "WARNING"

    def get_error_summary(self) -> Dict[str, Any]:
        """获取最近错误的摘要"""
        return {
            'total_errors': len(self.error_history),
            'error_types': self._count_error_types(),
            'recent_errors': [
                {
                    'type': e.error_type,
                    'message': e.message,
                    'timestamp': e.timestamp.isoformat()
                }
                for e in self.error_history[-5:]
            ]
        }

    def _count_error_types(self) -> Dict[str, int]:
        """统计每种错误类型的出现次数"""
        counts = {}
        for error in self.error_history:
            counts[error.error_type] =
counts.get(error.error_type, 0) + 1
        return counts

    def export_error_report(self, filepath: str):
        """导出错误历史到文件"""
        try:
            with open(filepath, 'w') as f:
                json.dump(
                    {
                        'error_summary':  self.get_error_summary(),
```

```
                        'full_history': [
                            {
                                'type': e.error_type,
                                'message': e.message,
                                'timestamp':
e.timestamp.isoformat(),
                                'severity': e.severity,
                                'context': {
                                    'component':
e.context.component,
                                    'function':
e.context.function,
                                    'line':  e.context.line_number,
                                    'file': e.context.file_path,
                                    'stack_trace':
e.context.stack_trace
                                }
                            }
                            for e in self.error_history
                        ]
                    },
                    f,
                    indent=2
                )
        except Exception as e:
            self.logger.error(f"Failed to export error report:
{e}")

def error_boundary(fallback_component:
Optional[Callable[[Exception], Component]] = None):
    """错误边界装饰器"""
    def decorator(component_class):
        original_render = component_class.render

        @wraps(original_render)
        def wrapped_render(self, *args, **kwargs):
            boundary = ErrorBoundary(
                children=[original_render(self, *args, **kwargs)],
                fallback=fallback_component
```

```
            )
            return boundary.render()

        component_class.render = wrapped_render
        return component_class

    return decorator

class Button(Component):
    """预构建的 Button 组件"""

    def __init__(self, text: str, on_click: Optional[Callable] =
None):
        super().__init__()
        self.tag_name = "button"
        self.set_prop('text', text)
        if on_click:
            self.set_prop('on_click', on_click)

    def render(self) -> Element:
        button = Element(self.tag_name, text=self.props['text'])
        if 'on_click' in self.props:
            button.on('click', self.props['on_click'])
        return button

class Container(Component):
    """预构建的 Container 组件"""

    def __init__(self, *children: Component):
        super().__init__()
        for child in children:
            self.add_child(child)

    def render(self) -> Element:
        container = Element(self.tag_name)
        for child in self.children:
            container.add(child.render())
        return container
```

```
class Input(Component):
    """预构建的 Input 组件"""

    def __init__(self, placeholder: str = "", value: str = "",
on_change: Optional[Callable] = None):
        super().__init__()
        self.tag_name = "input"
        self.set_prop('placeholder', placeholder)
        self.set_prop('value', value)
        if on_change:
            self.set_prop('on_change', on_change)

    def render(self) -> Element:
        input_elem = Element(self.tag_name)
        input_elem.set_attr('placeholder',
self.props['placeholder'])
        input_elem.set_attr('value', self.props['value'])
        if 'on_change' in self.props:
            input_elem.on('change', self.props['on_change'])
        return input_elem

class Form(Component):
    """预构建的 Form 组件"""

    def __init__(self, on_submit: Optional[Callable] = None):
        super().__init__()
        self.tag_name = "form"
        if on_submit:
            self.set_prop('on_submit', on_submit)

    def render(self) -> Element:
        form = Element(self.tag_name)
        if 'on_submit' in self.props:
            form.on('submit', self.props['on_submit'])
        for child in self.children:
            form.add(child.render())
        return form

class Text(Component):
```

```
    """文本组件"""

    def __init__(self, text: str, tag: str = "span"):
        super().__init__()
        self.tag_name = tag
        self.set_prop('text', text)

    def render(self) -> Element:
        return Element(self.tag_name, text=self.text)

class Image(Component):
    """图像组件"""

    def __init__(self, src: str, alt: str = "", width: str = "",
height: str = ""):
        super().__init__()
        self.tag_name = "img"
        self.set_prop('src', src)
        self.set_prop('alt', alt)
        if width:
            self.set_prop('width', width)
        if height:
            self.set_prop('height', height)

    def render(self) -> Element:
        img = Element(self.tag_name)
        img.set_attr('src', self.src)
        img.set_attr('alt', self.alt)
        if 'width' in self.props:
            img.set_attr('width', self.width)
        if 'height' in self.props:
            img.set_attr('height', self.height)
        return img

class Link(Component):
    """链接组件"""

    def __init__(self, href: str, text: str = "", target: str =
```

```
"_self"):
        super().__init__()
        self.tag_name = "a"
        self.set_prop('href', href)
        self.set_prop('text', text)
        self.set_prop('target', target)

    def render(self) -> Element:
        link = Element(self.tag_name, text=self.text)
        link.set_attr('href', self.href)
        link.set_attr('target', self.target)
        return link

class List(Component):
    """列表组件"""

    def __init__(self, items: list[str] | None = None, ordered:
bool = False):
        super().__init__()
        self.tag_name = "ol" if ordered else "ul"
        self.set_prop('items', items or [])

    def add_item(self, item: str):
        if 'items' not in self.props:
            self.props['items'] = []
        self.props['items'].append(item)

    def render(self) -> Element:
        list_elem = Element(self.tag_name)
        for item in self.props.get('items', []):
            li = Element('li', text=str(item))
            list_elem.add(li)
        return list_elem

class Card(Component):
    """卡片组件"""

    def __init__(self, title: str = "", body: str = "", footer:
str = ""):
```

```
        super().__init__()
        self.tag_name = "div"
        self.set_prop('title', title)
        self.set_prop('body', body)
        self.set_prop('footer', footer)

    def render(self) -> Element:
        card = Element(self.tag_name)
        card.add_class('card')

        if self.title:
            header = Element('div')
            header.add_class('card-header')
            header.add(Element('h3', text=self.title))
            card.add(header)

        body = Element('div')
        body.add_class('card-body')
        body.add(Element('p', text=self.body))
        card.add(body)

        if self.footer:
            footer = Element('div')
            footer.add_class('card-footer')
            footer.add(Element('p', text=self.footer))
            card.add(footer)

        return card

class Grid(Component):
    """网格布局组件"""

    def __init__(self, columns: int = 12, gap: str = "1rem"):
        super().__init__()
        self.tag_name = "div"
        self.set_prop('columns', columns)
        self.set_prop('gap', gap)
        self.style.add(
            display="grid",
```

```
            grid_template_columns=f"repeat({columns}, 1fr)",
            gap=gap
        )

    def add_item(self, component: Component, column_span: int =
1):
        component.style.add(grid_column=f"span {column_span}")
        self.add_child(component)

    def render(self) -> Element:
        grid = Element(self.tag_name)
        for child in self.children:
            grid.add(child.render())
        return grid

class Select(Component):
    """选择组件"""

    def __init__(self, options: OptionsType, value: str = "",
on_change: Optional[Callable] = None):
        super().__init__()
        self.tag_name = "select"
        self.set_prop('options', options)
        self.set_prop('value', value)
        if on_change:
            self.set_prop('on_change', on_change)

    def render(self) -> Element:
        select = Element(self.tag_name)
        if 'on_change' in self.props:
            select.on('change', self.on_change)

        for option in self.options:
            opt = Element('option')
            opt.set_attr('value', option.get('value', ''))
            if option.get('value') == self.value:
                opt.set_attr('selected', 'selected')
            opt.text = option.get('label', option.get('value',
''))
```

```
            select.add(opt)

        return select

class Checkbox(Component):
    """复选框组件"""

    def __init__(self, label: str = "", checked: bool = False,
on_change: Optional[Callable] = None):
        super().__init__()
        self.tag_name = "input"
        self.set_prop('type', 'checkbox')
        self.set_prop('label', label)
        self.set_prop('checked', checked)
        if on_change:
            self.set_prop('on_change', on_change)

    def render(self) -> Element:
        container = Element('div')

        input_elem = Element(self.tag_name)
        input_elem.set_attr('type', 'checkbox')
        if self.checked:
            input_elem.set_attr('checked', 'checked')
        if 'on_change' in self.props:
            input_elem.on('change', self.on_change)
        container.add(input_elem)

        if self.label:
            label = Element('label')
            label.text = self.label
            container.add(label)

        return container

class Radio(Component):
    """单选框组件"""

    def __init__(self, name: str, value: str, label: str = "",
```

```
checked: bool = False, on_change: Optional[Callable] = None):
        super().__init__()
        self.tag_name = "input"
        self.set_prop('type', 'radio')
        self.set_prop('name', name)
        self.set_prop('value', value)
        self.set_prop('label', label)
        self.set_prop('checked', checked)
        if on_change:
            self.set_prop('on_change', on_change)

    def render(self) -> Element:
        container = Element('div')

        input_elem = Element(self.tag_name)
        input_elem.set_attr('type', 'radio')
        input_elem.set_attr('name', self.name)
        input_elem.set_attr('value', self.value)
        if self.checked:
            input_elem.set_attr('checked', 'checked')
        if 'on_change' in self.props:
            input_elem.on('change', self.on_change)
        container.add(input_elem)

        if self.label:
            label = Element('label')
            label.text = self.label
            container.add(label)

        return container

class TextArea(Component):
    """文本域组件"""

    def __init__(self, value: str = "", placeholder: str = "",
rows: int = 3, on_change: Optional[Callable] = None):
        super().__init__()
        self.tag_name = "textarea"
        self.set_prop('value', value)
```

```
        self.set_prop('placeholder', placeholder)
        self.set_prop('rows', rows)
        if on_change:
            self.set_prop('on_change', on_change)

    def render(self) -> Element:
        textarea = Element(self.tag_name, text=self.value)
        textarea.set_attr('placeholder', self.placeholder)
        textarea.set_attr('rows', str(self.rows))
        if 'on_change' in self.props:
            textarea.on('change', self.on_change)
        return textarea

class Navbar(Component):
    """导航栏组件"""

    def __init__(self, brand: str = "", items: list[dict[str,
str]] = None, theme: str = "light"):
        super().__init__()
        self.tag_name = "nav"
        self.set_prop('brand', brand)
        self.set_prop('items', items or [])
        self.set_prop('theme', theme)
        self.style.add(
            display="flex",
            align_items="center",
            padding="1rem",
            background_color="#ffffff" if theme == "light" else
"#343a40",
            color="#000000" if theme == "light" else "#ffffff"
        )

    def add_item(self, text: str, href: str = "#", active: bool =
False):
        self.props['items'].append({
            'text': text,
            'href': href,
            'active': active
        })
```

```
    def render(self) -> Element:
        nav = Element(self.tag_name)

        if self.brand:
            brand = Element('a')
            brand.add_class('navbar-brand')
            brand.set_attr('href', '#')
            brand.text = self.brand
            brand.style.add(
                font_size="1.25rem",
                padding_right="1rem",
                text_decoration="none",
                color="inherit"
            )
            nav.add(brand)

        items_container = Element('div')
        items_container.add_class('navbar-items')
        items_container.style.add(
            display="flex",
            gap="1rem"
        )

        for item in self.items:
            link = Element('a')
            link.set_attr('href', item.get('href', '#'))
            link.text = item.get('text', '')
            link.style.add(
                text_decoration="none",
                color="inherit"
            )
            if item.get('active'):
                link.style.add(font_weight="bold")
            items_container.add(link)

        nav.add(items_container)
        return nav

class Flex(Component):
```

```
    """Flexbox 容器组件"""

    def __init__(self, direction: str = "row", justify: str =
"flex-start", align: str = "stretch", wrap: bool = False, gap:
str = "0"):
        super().__init__()
        self.tag_name = "div"
        self.style.add(
            display="flex",
            flex_direction=direction,
            justify_content=justify,
            align_items=align,
            flex_wrap="wrap" if wrap else "nowrap",
            gap=gap
        )

    def render(self) -> Element:
        flex = Element(self.tag_name)
        for child in self.children:
            flex.add(child.render())
        return flex

class ModernModal(Component):
    """现代模态对话框组件"""
    def __init__(self,
                 title: str,
                 content: str,
                 size: Literal["sm", "md", "lg", "xl"] = "md",
                 centered: bool = True,
                 closable: bool = True):
        super().__init__()
        self.tag_name = "div"
        self.set_prop('title', title)
        self.set_prop('content', content)
        self.set_prop('size', size)
        self.set_prop('centered', centered)
        self.set_prop('closable', closable)

        self.state.update({
```

```
            'visible': False
        })

        # 设置样式
        self.style.add(
            position="fixed",
            top="0",
            left="0",
            width="100%",
            height="100%",
            display="flex",
            align_items="center" if centered else "flex-start",
            justify_content="center",
            background_color="rgba(0, 0, 0, 0.5)",
            z_index="1000",
            opacity="0",
            visibility="hidden",
            transition="opacity 0.3s ease-in-out, visibility 0.3s
ease-in-out"
        )

    def show(self) -> None:
        """显示模态对话框"""
        self.set_state('visible', True)
        self.style.add(
            opacity="1",
            visibility="visible"
        )

    def hide(self) -> None:
        """隐藏模态对话框"""
        self.set_state('visible', False)
        self.style.add(
            opacity="0",
            visibility="hidden"
        )

    def _get_size_width(self) -> str:
```

```
        """Get modal width based on size"""
        size_map = {
            'sm': '300px',
            'md': '500px',
            'lg': '800px',
            'xl': '1140px'
        }
        return size_map.get(self.props['size'], '500px')

    def render(self):
        """渲染模态对话框"""
        dialog = Component()
        dialog.tag_name = "div"
        dialog.style.add(
            background_color="#ffffff",
            border_radius="0.5rem",
            box_shadow="0 25px 50px -12px rgba(0, 0, 0, 0.25)",
            max_width=self._get_size_width(),
            width="100%",
            max_height="90vh",
            display="flex",
            flex_direction="column",
            transform=f"scale({1 if self.state['visible'] else
0.9})",
            transition="transform 0.3s ease-in-out"
        )

        # Header
        header = Component()
        header.tag_name = "div"
        header.style.add(
            padding="1rem",
            border_bottom="1px solid #e5e7eb",
            display="flex",
            align_items="center",
            justify_content="space-between"
        )

        title = Component()
```

```
title.tag_name = "h3"
title.style.add(
    margin="0",
    font_size="1.25rem",
    font_weight="600",
    color="#111827"
)
title.set_text(self.props['title'])
header.add_child(title)

if self.props['closable']:
    close_button = Component()
    close_button.tag_name = "button"
    close_button.style.add(
        background="none",
        border="none",
        padding="0.5rem",
        cursor="pointer",
        color="#6b7280"
    )
    close_button.set_text("×")
    close_button.on_click.add(self.hide)
    header.add_child(close_button)

dialog.add_child(header)

# Content
content = Component()
content.tag_name = "div"
content.style.add(
    padding="1rem",
    overflow_y="auto"
)

if isinstance(self.props['content'], str):
    content.set_text(self.props['content'])
else:
    content.add_child(self.props['content'])

dialog.add_child(content)
```

```
        return dialog

class ModernToast(Component):
    """现代吐司通知组件"""

    def __init__(self,
                 message: str,
                 type: str = "info",
                 duration: int = 3000,
                 position: str = "bottom-right"):
        super().__init__()
        self.tag_name = "div"
        self.set_prop('message', message)
        self.set_prop('type', type)
        self.set_prop('duration', duration)
        self.set_prop('position', position)

        self.state.update({
            'visible': False
        })

        # 设置样式
        self.style.add(
            position="fixed",
            padding="1rem",
            border_radius="0.5rem",
            background_color=self._get_background_color(),
            color="#ffffff",
            box_shadow="0 10px 15px -3px rgba(0, 0, 0, 0.1)",
            max_width="24rem",
            opacity="0",
            transform="translateY(1rem)",
            transition="opacity 0.3s ease-in-out, transform 0.3s
ease-in-out",
            **self._get_position_style()
        )

    def show(self):
```

```
    """显示吐司通知"""
    self.set_state('visible', True)
    self.style.add(
        opacity="1",
        transform="translateY(0)"
    )

    # Auto hide
    if self.props['duration'] > 0:
        def hide():
            self.hide()
        setTimeout(hide, self.props['duration'])

def hide(self):
    """隐藏吐司通知"""
    self.set_state('visible', False)
    self.style.add(
        opacity="0",
        transform="translateY(1rem)"
    )

def _get_background_color(self) -> str:
    """Get background color based on type"""
    colors = {
        "info": "#3b82f6",
        "success": "#10b981",
        "warning": "#f59e0b",
        "error": "#ef4444"
    }
    return colors.get(self.props['type'], colors['info'])

def _get_position_style(self) -> dict[str, str]:
    """Get position style"""
    positions = {
        "top-left": {"top": "1rem", "left": "1rem"},
        "top-right": {"top": "1rem", "right": "1rem"},
        "bottom-left": {"bottom": "1rem", "left": "1rem"},
        "bottom-right": {"bottom": "1rem", "right": "1rem"}
    }
```

```
        return positions.get(self.props['position'],
positions['bottom-right'])

    def render(self):
        """Render toast"""
        container = Component()
        container.tag_name = "div"
        container.style.add(
            display="flex",
            align_items="center",
            gap="0.5rem"
        )

        # Icon
        icon = Component()
        icon.tag_name = "span"
        icon.style.add(
            font_size="1.25rem"
        )
        icon.set_text(self._get_icon())
        container.add_child(icon)

        # Message
        message = Component()
        message.tag_name = "span"
        message.set_text(self.props['message'])
        container.add_child(message)

        return container

    def _get_icon(self) -> str:
        """Get icon based on type"""
        icons = {
            "info": "ⓘ",
            "success": "✓",
            "warning": "⚠",
            "error": "✕"
        }
        return icons.get(self.props['type'], icons['info'])
```

```
class ModernTabs(Component):
    """现代选项卡组件"""

    def __init__(self,
                 tabs: list[dict[str, Any]],
                 active_index: int = 0,
                 variant: str = "default"):
        super().__init__()
        self.tag_name = "div"
        self.set_prop('tabs', tabs)
        self.set_prop('variant', variant)

        self.state.update({
            'active_index': active_index
        })

    def _handle_tab_click(self, index: int):
        """Handle tab click"""
        self.set_state('active_index', index)

    def render(self):
        """Render tabs"""
        container = Component()
        container.tag_name = "div"

        # Tab list
        tab_list = Component()
        tab_list.tag_name = "div"
        tab_list.style.add(
            display="flex",
            border_bottom="1px solid #e5e7eb"
        )

        for i, tab in enumerate(self.props['tabs']):
            tab_button = Component()
            tab_button.tag_name = "button"
            tab_button.style.add(
                padding="0.75rem 1rem",
```

```
                border="none",
                background="none",
                font_weight="500",
                color="#6b7280" if i != self.state['active_index']
else "#111827",
                border_bottom=f"2px solid {'transparent' if i !=
self.state['active_index'] else '#3b82f6'}",
                cursor="pointer",
                transition="all 0.2s ease-in-out"
            )
            tab_button.set_text(tab['label'])
            tab_button.on_click.add(lambda e, i=i:
self._handle_tab_click(i))
            tab_list.add_child(tab_button)

        container.add_child(tab_list)

        # Tab panels
        panel_container = Component()
        panel_container.tag_name = "div"
        panel_container.style.add(
            padding="1rem"
        )

        active_tab =
self.props['tabs'][self.state['active_index']]
        if isinstance(active_tab['content'], str):
            panel_container.set_text(active_tab['content'])
        else:
            panel_container.add_child(active_tab['content'])

        container.add_child(panel_container)

        return container

class ModernAccordion(Component):
    """现代手风琴组件"""

    def __init__(self,
```

```
                 items: list[dict[str, Any]],
                 multiple: bool = False):
        super().__init__()
        self.tag_name = "div"
        self.set_prop('items', items)
        self.set_prop('multiple', multiple)

        self.state.update({
            'expanded': set()
        })

    def _toggle_item(self, index: int):
        """Toggle accordion item"""
        expanded = self.state['expanded'].copy()

        if not self.props['multiple']:
            expanded.clear()

        if index in expanded:
            expanded.remove(index)
        else:
            expanded.add(index)

        self.set_state('expanded', expanded)

    def render(self):
        """Render accordion"""
        container = Component()
        container.tag_name = "div"
        container.style.add(
            border="1px solid #e5e7eb",
            border_radius="0.5rem",
            overflow="hidden"
        )

        for i, item in enumerate(self.props['items']):
            # Item container
            item_container = Component()
            item_container.tag_name = "div"
            item_container.style.add(
```

```
                border_top="1px solid #e5e7eb" if i > 0 else
"none"
            )

            # Header
            header = Component()
            header.tag_name = "button"
            header.style.add(
                width="100%",
                padding="1rem",
                background="none",
                border="none",
                text_align="left",
                cursor="pointer",
                display="flex",
                align_items="center",
                justify_content="space-between"
            )

            # Expand/collapse icon
            has_children = 'children' in item and item['children']
            if has_children:
                icon = Component()
                icon.tag_name = "span"
                icon.style.add(
                    margin_right="0.5rem",
                    transition="transform 0.2s"
                )
                if i in self.state['expanded']:
                    icon.style.add(transform="rotate(90deg)")
                icon.add(Element('span', text="▶"))
                header.add(icon)

            # Node icon (if provided)
            if 'icon' in item:
                node_icon = Component()
                node_icon.tag_name = "span"
                node_icon.style.add(margin_right="0.5rem")
                node_icon.add(Element('span', text=item['icon']))
                header.add(node_icon)
```

```
            # Node label
            label = Component()
            label.tag_name = "span"
            label.add(Element('span', text=item['label']))
            header.add(label)

            # Add click handler for expansion toggle
            if has_children:
                header.on('click', lambda: self._toggle_item(i))

            item_container.add(header)

            # Render children if node is expanded
            if has_children and i in self.state['expanded']:
                children_container = Component()
                for child in item['children']:

children_container.add(self._render_node(child, 1))
                item_container.add(children_container)

            container.add_child(item_container)

        return container

    def _render_node(self, node: Dict[str, Any], level: int = 0)
-> Element:
        """Render a single node and its children"""
        node_container = Element('div')

        # Node header
        header = Element('div')
        header.style.add(
            display="flex",
            align_items="center",
            padding="0.5rem",
            padding_left=f"{level * 1.5 + 0.5}rem",
            cursor="pointer",
            transition="background-color 0.2s"
        )
```

```
        header.add_hover_style(background_color="#f5f5f5")

        # Expand/collapse icon
        has_children = 'children' in node and node['children']
        if has_children:
            icon = Element('span')
            icon.style.add(
                margin_right="0.5rem",
                transition="transform 0.2s"
            )
            if node['id'] in self.state['expanded']:
                icon.style.add(transform="rotate(90deg)")
            icon.add(Element('span', text="▶"))
            header.add(icon)

        # Node icon (if provided)
        if 'icon' in node:
            node_icon = Element('span')
            node_icon.style.add(margin_right="0.5rem")
            node_icon.add(Element('span', text=node['icon']))
            header.add(node_icon)

        # Node label
        label = Element('span')
        label.add(Element('span', text=node['label']))
        header.add(label)

        # Add click handler for expansion toggle
        if has_children:
            header.on('click', lambda:
self._toggle_item(node['id']))

        node_container.add(header)

        # Render children if node is expanded
        if has_children and node['id'] in self.state['expanded']:
            children_container = Element('div')
            for child in node['children']:
                children_container.add(self._render_node(child,
level + 1))
```

```
            node_container.add(children_container)

        return node_container

class VirtualList(Component):
    """虚拟滚动列表组件，用于高效渲染大量数据"""

    def __init__(self,
                 items: List[Any],
                 render_item: Callable[[Any], Component],
                 item_height: int = 40,
                 container_height: int = 400,
                 buffer_size: int = 5):
        super().__init__()
        self.tag_name = "div"
        self.set_prop('items', items)
        self.set_prop('render_item', render_item)
        self.set_prop('item_height', item_height)
        self.set_prop('container_height', container_height)
        self.set_prop('buffer_size', buffer_size)

        self.state.update({
            'scroll_top': 0,
            'visible_items': [],
            'total_height': len(items) * item_height,
            'padding_top': 0,
            'padding_bottom': 0
        })

        self.style.add(
            height=f"{container_height}px",
            overflow_y="auto",
            position="relative"
        )

        self.on_scroll = EventDelegate()
        self.on_scroll.add(self._handle_scroll)

    def _handle_scroll(self, event: Dict[str, Any]):
```

```
        """处理滚动事件"""
        scroll_top = event['target'].scrollTop
        self._update_visible_items(scroll_top)

    def _update_visible_items(self, scroll_top: int):
        """更新可见项目列表"""
        self.state['scroll_top'] = scroll_top

        # 计算可见范围
        start_index = max(0, scroll_top //
self.props['item_height'] - self.props['buffer_size'])
        visible_count = (self.props['container_height'] //
self.props['item_height'] +
                         2 * self.props['buffer_size'])
        end_index = min(len(self.props['items']), start_index +
visible_count)

        # 更新可见项目
        self.state['visible_items'] =
self.props['items'][start_index:end_index]

        # 更新 padding 以保持滚动位置
        self.state['padding_top'] = start_index *
self.props['item_height']
        self.state['padding_bottom'] = (
            (len(self.props['items']) - end_index) *
self.props['item_height']
        )

    def render(self):
        """渲染虚拟列表"""
        # 容器
        container = Component()
        container.tag_name = "div"
        container.style.add(
            height="100%",
            overflow_y="auto"
```

```
        )

        # 内容包装器
        content = Component()
        content.tag_name = "div"
        content.style.add(
            position="relative",
            height=f"{self.state['total_height']}px"
        )

        # 可见项目容器
        items_container = Component()
        items_container.tag_name = "div"
        items_container.style.add(
            position="absolute",
            top=f"{self.state['padding_top']}px",
            left="0",
            right="0"
        )

        # 渲染可见项目
        for item in self.state['visible_items']:
            rendered_item = self.props['render_item'](item)
            rendered_item.style.add(
                height=f"{self.props['item_height']}px"
            )
            items_container.add_child(rendered_item)

        content.add_child(items_container)
        container.add_child(content)

        return container

class DraggableList(Component):
    """可拖放的列表组件"""

    def __init__(self,
                 items: list[Any],
```

```
                 render_item: Optional[Callable[[Any], Component]]
= None,
                 on_reorder: Optional[Callable[[list[Any]], None]]
= None):
        super().__init__()
        self.tag_name = "div"
        self.set_prop('items', items)
        self.set_prop('render_item', render_item or
self._default_render_item)
        self.set_prop('on_reorder', on_reorder)

        self.state.update({
            'dragging_index': None,
            'drag_over_index': None,
            'items': items.copy()
        })

        # 设置容器样式
        self.style.add(
            position="relative",
            user_select="none"
        )

    def _default_render_item(self, item: Any) -> Component:
        """默认项渲染器"""
        text = Text(str(item))
        text.style.add(
            padding="1rem",
            background_color="#ffffff",
            border="1px solid #e0e0e0",
            margin_bottom="0.5rem",
            cursor="move"
        )
        return text

    def _handle_drag_start(self, index: int, event: dict[str,
Any]):
        """处理拖拽开始事件"""
```

```
        try:
            self.state['dragging_index'] = index
            self._update()
        except Exception as e:
            self._logger.error(f"Error handling drag start: {e}",
exc_info=True)

    def _handle_drag_over(self, index: int, event: dict[str,
Any]):
        """处理拖拽悬停事件"""
        try:
            if index != self.state['drag_over_index']:
                self.state['drag_over_index'] = index
                self._update()
        except Exception as e:
            self._logger.error(f"Error handling drag over: {e}",
exc_info=True)

    def _handle_drop(self, index: int, event: dict[str, Any]):
        """处理放置事件"""
        try:
            dragging_index = self.state['dragging_index']
            if dragging_index is not None and dragging_index !=
index:
                items = self.state['items']
                item = items.pop(dragging_index)
                items.insert(index, item)

                if self.props['on_reorder']:
                    self.props['on_reorder'](items)

            self.state.update({
                'dragging_index': None,
                'drag_over_index': None
            })
            self._update()

        except Exception as e:
            self._logger.error(f"Error handling drop: {e}",
```

```
exc_info=True)

    def render(self) -> Element:
        """渲染可拖放列表"""
        try:
            container = super().render()
            items = self.state['items']
            dragging_index = self.state['dragging_index']
            drag_over_index = self.state['drag_over_index']

            for i, item in enumerate(items):
                item_container = Element('div')
                item_container.style.add(
                    opacity="1" if i != dragging_index else "0.5",
                    transform="none" if i != drag_over_index else
"translateY(8px)",
                    transition="transform 0.15s ease-in-out"
                )

                # 添加拖放事件监听器
                item_container.set_attribute('draggable', 'true')
                item_container.add_event_listener('dragstart',
lambda e, i=i: self._handle_drag_start(i, e))
                item_container.add_event_listener('dragover',
lambda e, i=i: self._handle_drag_over(i, e))
                item_container.add_event_listener('drop', lambda
e, i=i: self._handle_drop(i, e))

                # 渲染项内容
                item_content = self.props['render_item'](item)
                item_container.append_child(item_content.render())

                container.append_child(item_container)

            return container

        except Exception as e:
            self._logger.error(f"Error rendering draggable list:
{e}", exc_info=True)
```

```
            raise

class Table(Component):
    """表格组件"""

    def __init__(self, columns: list[dict[str, str]], data:
list[dict[str, Any]],
                 sortable: bool = True, filterable: bool = True,
                 page_size: int = 10):
        super().__init__()
        self.tag_name = "div"
        self.set_prop('columns', columns)  # [{"key": "id",
"title": "ID"}, ...]
        self.set_prop('data', data)
        self.set_prop('sortable', sortable)
        self.set_prop('filterable', filterable)
        self.set_prop('page_size', page_size)
        self.set_prop('current_page', 1)

        # State for sorting and filtering
        self.state['sort_key'] = None
        self.state['sort_order'] = 'asc'
        self.state['filters'] = {}

    def render(self):
        container = Element('div')

        # Create table element
        table = Element('table')
        table.style.add(
            width="100%",
            border_collapse="collapse",
            margin="1rem 0"
        )

        # Render header
        header = Element('thead')
        header_row = Element('tr')
```

```
        for col in self.props['columns']:
            th = Element('th')
            th.style.add(
                padding="0.75rem",
                border_bottom="2px solid #ddd",
                text_align="left",
                font_weight="bold"
            )

            if self.props['sortable']:
                sort_container = Element('div')
                sort_container.style.add(
                    display="flex",
                    align_items="center",
                    cursor="pointer"
                )
                sort_container.add(Element('span',
text=col['title']))
                sort_container.add(Element('span', text="↕",
style={"margin-left": "0.5rem"}))
                th.add(sort_container)
            else:
                th.add(Element('span', text=col['title']))

            header_row.add(th)

        header.add(header_row)
        table.add(header)

        # Render body
        body = Element('tbody')

        # Apply pagination
        start_idx = (self.props['current_page'] - 1) *
self.props['page_size']
        end_idx = start_idx + self.props['page_size']
        page_data = self.props['data'][start_idx:end_idx]

        for row_data in page_data:
            tr = Element('tr')
```

```
            tr.style.add(
                border_bottom="1px solid #ddd",
                transition="background-color 0.2s"
            )
            tr.add_hover_style(background_color="#f5f5f5")

            for col in self.props['columns']:
                td = Element('td')
                td.style.add(padding="0.75rem")
                td.add(Element('span',
text=str(row_data.get(col['key'], ''))))
                tr.add(td)

            body.add(tr)

        table.add(body)
        container.add(table)

        # Add pagination
        if len(self.props['data']) > self.props['page_size']:
            pagination = self._render_pagination()
            container.add(pagination)

        return container

    def _render_pagination(self):
        total_pages = (len(self.props['data']) +
self.props['page_size'] - 1) // self.props['page_size']

        pagination = Element('div')
        pagination.style.add(
            display="flex",
            justify_content="center",
            align_items="center",
            margin_top="1rem"
        )

        # Previous button
        prev_btn = Element('button')
        prev_btn.add(Element('span', text="Previous"))
```

```
        prev_btn.style.add(
            padding="0.5rem 1rem",
            margin="0 0.25rem",
            border="1px solid #ddd",
            border_radius="4px",
            cursor="pointer" if self.props['current_page'] > 1
else "not-allowed",
            background_color="#fff"
        )
        pagination.add(prev_btn)

        # Page numbers
        for page in range(1, total_pages + 1):
            page_btn = Element('button')
            page_btn.add(Element('span', text=str(page)))
            page_btn.style.add(
                padding="0.5rem 1rem",
                margin="0 0.25rem",
                border="1px solid #ddd",
                border_radius="4px",
                cursor="pointer",
                background_color="#fff" if page !=
self.props['current_page'] else "#e6e6e6"
            )
            pagination.add(page_btn)

        # Next button
        next_btn = Element('button')
        next_btn.add(Element('span', text="Next"))
        next_btn.style.add(
            padding="0.5rem 1rem",
            margin="0 0.25rem",
            border="1px solid #ddd",
            border_radius="4px",
            cursor="pointer" if self.props['current_page'] <
total_pages else "not-allowed",
            background_color="#fff"
        )
        pagination.add(next_btn)
```

```
        return pagination

class Tree(Component):
    """树形组件"""
    def __init__(self, data: List[Dict[str, Any]], expanded: bool
= False):
        """
        初始化树形组件
        data: 树形数据，每个节点是一个字典，包含'id'、'label'、
'children'等键
        """
        super().__init__()
        self.tag_name = "div"
        self.set_prop('data', data)
        self.state['expanded'] = set()  # Store expanded node IDs

        # Expand all nodes if expanded is True
        if expanded:
            self._expand_all(data)

    def _expand_all(self, nodes: List[Dict[str, Any]]) -> None:
        """递归展开所有节点"""
        for node in nodes:
            self.state['expanded'].add(node['id'])
            if node.get('children'):
                self._expand_all(node['children'])

    def toggle_node(self, node_id: str) -> None:
        """Toggle node expansion state"""
        if node_id in self.state['expanded']:
            self.state['expanded'].remove(node_id)
        else:
            self.state['expanded'].add(node_id)
        self._update()

    def _render_node(self, node: Dict[str, Any], level: int = 0)
-> Element:
```

```
"""Render a single node and its children"""
node_container = Element('div')

# Node header
header = Element('div')
header.style.add(
    display="flex",
    align_items="center",
    padding="0.5rem",
    padding_left=f"{level * 1.5 + 0.5}rem",
    cursor="pointer",
    transition="background-color 0.2s"
)
header.add_hover_style(background_color="#f5f5f5")

# Expand/collapse icon
has_children = 'children' in node and node['children']
if has_children:
    icon = Element('span')
    icon.style.add(
        margin_right="0.5rem",
        transition="transform 0.2s"
    )
    if node['id'] in self.state['expanded']:
        icon.style.add(transform="rotate(90deg)")
    icon.add(Element('span', text="▶"))
    header.add(icon)

# Node icon (if provided)
if 'icon' in node:
    node_icon = Element('span')
    node_icon.style.add(margin_right="0.5rem")
    node_icon.add(Element('span', text=node['icon']))
    header.add(node_icon)

# Node label
label = Element('span')
label.add(Element('span', text=node['label']))
header.add(label)
```

```
        # Add click handler for expansion toggle
        if has_children:
            header.on('click', lambda:
self.toggle_node(node['id']))

        node_container.add(header)

        # Render children if node is expanded
        if has_children and node['id'] in self.state['expanded']:
            children_container = Element('div')
            for child in node['children']:
                children_container.add(self._render_node(child,
level + 1))
            node_container.add(children_container)

        return node_container

    def render(self):
        container = Element('div')
        container.style.add(
            border="1px solid #ddd",
            border_radius="4px",
            overflow="hidden"
        )

        # Render each root node
        for node in self.props['data']:
            container.add(self._render_node(node))

        return container

class Responsive(Component):
    """响应式容器组件"""
    breakpoints = {
        'sm': '576px',
        'md': '768px',
        'lg': '992px',
        'xl': '1200px',
        'xxl': '1400px'
```

```
    }

    def __init__(self):
        super().__init__()
        self.tag_name = "div"
        self.style.add(
            width="100%",
            margin="0 auto",
            padding="0 15px",
            box_sizing="border-box"
        )

    def add_media_query(self, breakpoint: str, styles: dict[str,
str]):
        self.style.add_media_query(
            f"(min-width: {self.breakpoints[breakpoint]})",
            styles
        )
        return self

class Skeleton(Component):
    """骨架屏组件"""
    def __init__(self, type: str = "text", rows: int = 1, height:
str = "1rem"):
        super().__init__()
        self.tag_name = "div"
        self.set_prop('type', type)
        self.set_prop('rows', rows)
        self.set_prop('height', height)
        self.style.add(
            background="linear-gradient(90deg, #f0f0f0 25%,
#e0e0e0 50%, #f0f0f0 75%)",
            background_size="200% 100%",
            animation="skeleton-loading 1.5s infinite",
            border_radius="4px",
            height=height,
            margin_bottom="0.5rem"
        )
```

```
class Carousel(Component):
    """幻灯片组件"""
    def __init__(self, images: list[dict[str, str]], auto_play:
bool = True, interval: int = 3000):
        super().__init__()
        self.tag_name = "div"
        self.set_prop('images', images)  # [{"src": "...", "alt":
"..."}]
        self.set_prop('auto_play', auto_play)
        self.set_prop('interval', interval)
        self.state['current_index'] = 0
        self.style.add(
            position="relative",
            overflow="hidden",
            width="100%",
            height="100%"
        )

class Drawer(Component):
    """抽屉组件"""
    def __init__(self, content: Component, position: str = "left",
width: str = "300px"):
        super().__init__()
        self.tag_name = "div"
        self.set_prop('content', content)
        self.set_prop('position', position)
        self.set_prop('width', width)
        self.state['visible'] = False
        self.style.add(
            position="fixed",
            top="0",
            height="100%",
            background_color="#ffffff",
            box_shadow="0 0 10px rgba(0,0,0,0.1)",
            transition="transform 0.3s ease-in-out",
            z_index="1000"
        )

class Progress(Component):
```

```
    """进度条组件"""
    def __init__(self, value: int = 0, max: int = 100, type: str
= "bar", color: str = "#007bff"):
        super().__init__()
        self.tag_name = "div"
        self.set_prop('value', value)
        self.set_prop('max', max)
        self.set_prop('type', type)
        self.set_prop('color', color)
        self.style.add(
            width="100%",
            height="0.5rem",
            background_color="#e9ecef",
            border_radius="0.25rem",
            overflow="hidden"
        )

class Badge(Component):
    """徽章组件"""
    def __init__(self, text: str, type: str = "primary", pill:
bool = False):
        super().__init__()
        self.tag_name = "span"
        self.set_prop('text', text)
        self.set_prop('type', type)
        self.set_prop('pill', pill)
        self.style.add(
            display="inline-block",
            padding="0.25em 0.4em",
            font_size="75%",
            font_weight="700",
            line_height="1",
            text_align="center",
            white_space="nowrap",
            vertical_align="baseline",
            border_radius="0.25rem" if not pill else "10rem",
            color="#fff",
            background_color=self._get_type_color(type)
        )
```

```
    def _get_type_color(self, type: str) -> str:
        colors = {
            'primary': '#007bff',
            'secondary': '#6c757d',
            'success': '#28a745',
            'danger': '#dc3545',
            'warning': '#ffc107',
            'info': '#17a2b8'
        }
        return colors.get(type, colors['primary'])

class Tooltip(Component):
    """提示框组件"""
    def __init__(self, content: str, position: str = "top"):
        super().__init__()
        self.tag_name = "div"
        self.set_prop('content', content)
        self.set_prop('position', position)
        self.style.add(
            position="relative",
            display="inline-block"
        )
```

config.py

```
import os
from dotenv import load_dotenv

basedir = os.path.abspath(os.path.dirname(__file__))
load_dotenv()

class Config:
    SECRET_KEY = os.environ.get('SECRET_KEY') or 'dev-key-please-
change-in-production'
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL') or \
        'sqlite:///' + os.path.join(basedir, 'app.db')
```

```
    SQLALCHEMY_TRACK_MODIFICATIONS = False
```

core.py

```
"""
Core functionality for PytoWeb framework
"""

from typing import Dict, Any, Optional, Type
from .components import Component
from .vdom import VNode, VDOMRenderer, VDOMDiffer
from .events import EventBridge, StateManager
from .styles import Style
from .themes import Theme, ThemeProvider
from .animations import AnimationManager

class PytoWeb:
    """Main PytoWeb framework class"""

    def __init__(self):
        self.state_manager = StateManager()
        self.event_bridge = EventBridge()
        self.theme = Theme()
        ThemeProvider.set_theme(self.theme)

    def create_app(self, root_component: Type[Component], props:
Dict[str, Any] = None) -> str:
        """Create a new PytoWeb application"""
        # Initialize root component
        instance = root_component()
        if props:
            for key, value in props.items():
                instance.set_prop(key, value)

        # Generate HTML
        html = self._generate_html(instance)

        # Add framework scripts
```

```
        html += self._get_framework_scripts()

        return html

    def _generate_html(self, root: Component) -> str:
        """Generate HTML from component tree"""
        # Get all registered animations CSS
        animations_css = AnimationManager.get_all_css()

        # Create HTML template
        html = f"""
        <!DOCTYPE html>
        <html>
        <head>
            <meta charset="UTF-8">
            <meta name="viewport" content="width=device-width,
initial-scale=1.0">
            <style>
                {animations_css}
                {self._get_default_styles()}
            </style>
        </head>
        <body>
            <div id="app">
                {root.render()}
            </div>
        </body>
        </html>
        """
        return html

    def _get_framework_scripts(self) -> str:
        """Get framework JavaScript code"""
        return f"""
        <script>
        {EventBridge.get_client_script()}

        // Framework initialization
        document.addEventListener('DOMContentLoaded', function()
{{
```

```
            console.log('PytoWeb initialized');
        }});
        </script>
        """

    def _get_default_styles(self) -> str:
        """Get default framework styles"""
        from .styles import DEFAULT_STYLES
        return DEFAULT_STYLES

    def handle_event(self, event_type: str, event_data: Dict[str,
Any]):
        """Handle framework events"""
        self.event_bridge.handle_event(event_type, event_data)

    def set_theme(self, theme: Theme):
        """Set framework theme"""
        self.theme = theme
        ThemeProvider.set_theme(theme)

    def get_state_manager(self) -> StateManager:
        """Get state manager instance"""
        return self.state_manager

    @staticmethod
    def create_style(**styles) -> Style:
        """Create a new style instance"""
        return Style(**styles)

    @staticmethod
    def register_animation(name: str, keyframes: Dict[str,
Dict[str, str]]):
        """Register a new animation"""
        from .animations import Animation
        animation = Animation(name, keyframes)
        AnimationManager.register(animation)
```

elements.py

```
"""
Core module for HTML element creation and manipulation
"""

from __future__ import annotations
from typing import List, Dict, Any, Optional, Callable, Union,
TypeVar, TYPE_CHECKING
from dataclasses import dataclass
from .styles import Style

T = TypeVar('T', bound='Element')

class ElementError(Exception):
    """Element creation or manipulation error"""
    pass

@dataclass
class EventHandler:
    """Event handler container"""
    name: str
    handler: Callable[..., Any]

    def __post_init__(self):
        if not callable(self.handler):
            raise ElementError("Event handler must be callable")

class Element:
    """Base class for all HTML elements"""

    VOID_ELEMENTS = {
        'area', 'base', 'br', 'col', 'embed', 'hr', 'img',
'input',
        'link', 'meta', 'param', 'source', 'track', 'wbr'
    }

    def __init__(self, tag: str, text: str = "", **attributes:
Any):
        """Initialize element"""
```

```
        try:
            if not isinstance(tag, str):
                raise ElementError("Tag must be a string")
            if not tag:
                raise ElementError("Tag cannot be empty")

            self.tag = tag.lower()  # Normalize tag name
            self.text = str(text)  # Ensure text is string
            self.attributes: Dict[str, Any] = {}
            self.children: List[Element] = []
            self.style = Style()  # Use Style class for style
management
            self.events: Dict[str, EventHandler] = {}

            # Process attributes
            for key, value in attributes.items():
                if value is not None:  # Only add non-None
attributes
                    self.attributes[key] = str(value)

        except Exception as e:
            raise ElementError(f"Failed to initialize element:
{e}") from e

    def add(self, child: Union[Element, str]) -> T:
        """Add child element"""
        try:
            if isinstance(child, str):
                child = Element('span', text=child)
            elif not isinstance(child, Element):
                raise ElementError("Child must be an Element or
string")

            self.children.append(child)
            return self

        except Exception as e:
            raise ElementError(f"Failed to add child: {e}") from
e
```

```
    def add_child(self, child: Union[Element, str]) -> T:
        """Alias for add()"""
        return self.add(child)

    def to_html(self) -> str:
        """Convert element to HTML string"""
        try:
            print(f"[DEBUG] Converting {self.tag} to HTML")
            # Start tag
            html = [f"<{self.tag}"]

            # Add attributes
            for key, value in self.attributes.items():
                html.append(f' {key}="{value}"')

            # Add style
            if self.style and self.style.get_all():
                style_str = '; '.join(f"{k}: {v}" for k, v in
self.style.get_all().items())
                html.append(f' style="{style_str}"')

            # Close start tag
            html.append('>')

            # Add text content
            if self.text:
                html.append(self.text)

            # Add children
            for child in self.children:
                try:
                    child_html = child.to_html()
                    html.append(child_html)
                except Exception as e:
                    print(f"[ERROR] Failed to render child of
{self.tag}: {e}")
                    raise

            # Add closing tag if not void element
            if self.tag not in self.VOID_ELEMENTS:
```

```
                html.append(f"</{self.tag}>")

            result = ''.join(html)
            print(f"[DEBUG] Generated HTML for {self.tag}:
{result[:100]}...")
            return result

        except Exception as e:
            print(f"[ERROR] Failed to generate HTML for
{self.tag}: {e}")
            raise ElementError(f"Failed to generate HTML: {e}")
from e

    def __str__(self) -> str:
        """String representation is HTML"""
        return self.to_html()

# Convenience functions for creating common elements
def div(*children: Element, **attrs: Any) -> Element:
    """Create a div element"""
    return Element('div', **attrs).add(*children)

def span(*children: Element, **attrs: Any) -> Element:
    """Create a span element"""
    return Element('span', **attrs).add(*children)

def p(*children: Element, **attrs: Any) -> Element:
    """Create a paragraph element"""
    return Element('p', **attrs).add(*children)

def a(href: str, *children: Element, **attrs: Any) -> Element:
    """Create an anchor element"""
    attrs['href'] = href
    return Element('a', **attrs).add(*children)

def img(src: str, alt: str = "", **attrs: Any) -> Element:
    """Create an image element"""
    attrs.update({'src': src, 'alt': alt})
    return Element('img', **attrs)
```

```
def button(*children: Element, **attrs: Any) -> Element:
    """Create a button element"""
    return Element('button', **attrs).add(*children)

def input(type: str = "text", **attrs: Any) -> Element:
    """Create an input element"""
    attrs['type'] = type
    return Element('input', **attrs)
```

events.py

```
from __future__ import annotations
from .state import StateManager

"""
PytoWeb 事件系统

提供事件处理、委托、批处理和状态管理功能。
"""

from typing import (
    Dict, Any, Optional, Callable, List, Set,
    TypeVar, TypedDict, Union
)
from collections import defaultdict
import asyncio
import time
import uuid
import json
import weakref
import logging
from concurrent.futures import ThreadPoolExecutor

# 配置日志
logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger(__name__)
```

```
# 类型别名
T = TypeVar('T')
EventType = str
HandlerType = Callable[..., None]
EventData = Dict[str, Any]

class Event:
    """事件基类"""
    def __init__(self, event_type: EventType, target: Any, data:
EventData | None = None):
        self.type = event_type
        self.target = target
        self.data = data or {}
        self.timestamp = time.time()
        self.propagation_stopped = False
        self.default_prevented = False

    def stop_propagation(self):
        """停止事件传播"""
        self.propagation_stopped = True

    def prevent_default(self):
        """阻止默认行为"""
        self.default_prevented = True

class EventHandler:
    """事件处理器"""
    def __init__(self,
                 callback: Callable[[Event], None],
                 once: bool = False,
                 capture: bool = False,
                 passive: bool = False):
        self.callback = callback
        self.once = once
        self.capture = capture
        self.passive = passive
```

```
class EventBridge:
    """Python 和 JavaScript 事件桥接器"""

    _handlers: Dict[str, Callable] = {}
    _js_code = """
    window.pytoweb = {
        handlers: {},

        handleEvent: function(handlerId, event) {
            // 发送事件数据到 Python
            const eventData = {
                type: event.type,
                target: {
                    id: event.target.id,
                    value: event.target.value,
                    checked: event.target.checked,
                    dataset: event.target.dataset,
                    scrollTop: event.target.scrollTop,
                    scrollHeight: event.target.scrollHeight,
                    clientHeight: event.target.clientHeight
                },
                clientX: event.clientX,
                clientY: event.clientY,
                timestamp: Date.now()
            };

            // 发送到 Python 后端
            this.sendToPython(handlerId, eventData);
        },

        sendToPython: async function(handlerId, data) {
            try {
                const response = await fetch('/api/events', {
                    method: 'POST',
                    headers: {
                        'Content-Type': 'application/json'
                    },
                    body: JSON.stringify({
```

```
                        handlerId: handlerId,
                        data: data
                    })
                });

                if (!response.ok) {
                    throw new Error('Event handling failed');
                }
            } catch (error) {
                console.error('Error sending event to Python:',
error);
            }
        },

        registerHandler: function(handlerId, options = {}) {
            const handler = (event) => {
                if (options.preventDefault) {
                    event.preventDefault();
                }
                if (options.stopPropagation) {
                    event.stopPropagation();
                }
                this.handleEvent(handlerId, event);
            };

            this.handlers[handlerId] = handler;
            return handler;
        },

        removeHandler: function(handlerId) {
            delete this.handlers[handlerId];
        }
    };
    """

    @classmethod
    def register_handler(cls, handler: Callable) -> str:
        """注册事件处理器并返回处理器 ID"""
        handler_id = str(uuid.uuid4())
```

```
        cls._handlers[handler_id] = handler
        return handler_id

    @classmethod
    def remove_handler(cls, handler_id: str):
        """移除事件处理器"""
        if handler_id in cls._handlers:
            del cls._handlers[handler_id]

    @classmethod
    def handle_event(cls, handler_id: str, event_data: Dict[str,
Any]):
        """处理从 JavaScript 发来的事件"""
        if handler_id in cls._handlers:
            try:
                event = Event(
                    event_type=event_data.get('type', ''),
                    target=event_data.get('target', {}),
                    data=event_data
                )
                cls._handlers[handler_id](event)
            except Exception as e:
                logger.error(f"Error handling event: {e}",
exc_info=True)

class EventDelegate:
    """事件委托类"""
    def __init__(self):
        self._handlers: list[HandlerType] = []
        self._logger = logging.getLogger(__name__)

    def add(self, handler: HandlerType):
        """添加事件处理器"""
        if handler not in self._handlers:
            self._handlers.append(handler)

    def remove(self, handler: HandlerType):
```

```
        """移除事件处理器"""
        if handler in self._handlers:
            self._handlers.remove(handler)

    def clear(self):
        """清除所有处理器"""
        self._handlers.clear()

    def __call__(self, *args, **kwargs):
        """调用所有处理器"""
        for handler in self._handlers:
            try:
                handler(*args, **kwargs)
            except Exception as e:
                self._logger.error(f"Error in event handler: {e}",
exc_info=True)

class EventEmitter:
    """增强的事件发射器，支持事件委托和批处理"""

    def __init__(self):
        self._handlers: dict[str, list[EventHandler]] =
defaultdict(list)
        self._delegate_handlers: dict[str, dict[str,
list[EventHandler]]] = defaultdict(lambda: defaultdict(list))
        self._batch_handlers: dict[str,
list[Callable[[list[Event]], None]]] = defaultdict(list)
        self._batch_queue: dict[str, list[Event]] =
defaultdict(list)
        self._batch_timeout = 16.67  # ~60fps
        self._logger = logging.getLogger(__name__)

    def on(self, event_type: str, callback: Callable[[Event],
None], selector: str | None = None, **options):
        """添加事件监听器"""
        handler = EventHandler(callback, **options)
```

```
        if selector:

self._delegate_handlers[event_type][selector].append(handler)
        else:
            self._handlers[event_type].append(handler)

    def off(self, event_type: str, callback: Callable[[Event],
None] | None = None, selector: str | None = None):
        """移除事件监听器"""
        if selector:
            if callback:
                self._delegate_handlers[event_type][selector] = [
                    h for h in
self._delegate_handlers[event_type][selector]
                    if h.callback != callback
                ]
            else:

self._delegate_handlers[event_type][selector].clear()
        else:
            if callback:
                self._handlers[event_type] = [
                    h for h in self._handlers[event_type]
                    if h.callback != callback
                ]
            else:
                self._handlers[event_type].clear()

    def emit(self, event: Event):
        """发射事件"""
        if event.propagation_stopped:
            return

        # 处理直接监听器
        for handler in self._handlers[event.type]:
            try:
                if handler.once:
                    self.off(event.type, handler.callback)
```

```
                handler.callback(event)
            except Exception as e:
                self._logger.error(f"Error in event handler: {e}",
exc_info=True)

        # 处理委托监听器
        if isinstance(event.target, dict) and 'id' in
event.target:
            target_id = event.target['id']
            for selector, handlers in
self._delegate_handlers[event.type].items():
                if self._matches_selector(target_id, selector):
                    for handler in handlers:
                        try:
                            if handler.once:
                                self.off(event.type,
handler.callback, selector)
                            handler.callback(event)
                        except Exception as e:
                            self._logger.error(f"Error in
delegate handler: {e}", exc_info=True)

        # 添加到批处理队列
        if event.type in self._batch_handlers:
            self._batch_queue[event.type].append(event)
            self._schedule_batch_process(event.type)

    def _matches_selector(self, target_id: str, selector: str) ->
bool:
        """检查目标是否匹配选择器"""
        # 简单的选择器匹配实现
        return selector.startswith('#') and target_id ==
selector[1:]

    def add_batch_handler(self, event_type: str, handler:
Callable[[list[Event]], None]):
        """添加批处理事件处理器"""
```

```
        self._batch_handlers[event_type].append(handler)

    def _schedule_batch_process(self, event_type: str):
        """调度批处理"""
        async def process_batch():
            await asyncio.sleep(self._batch_timeout / 1000)
            events = self._batch_queue[event_type]
            self._batch_queue[event_type] = []

            for handler in self._batch_handlers[event_type]:
                try:
                    handler(events)
                except Exception as e:
                    self._logger.error(f"Error in batch handler:
{e}", exc_info=True)

        asyncio.create_task(process_batch())

class EventManager:
    """全局事件管理系统"""

    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance

    def __init__(self):
        if not hasattr(self, 'initialized'):
            self.emitter = EventEmitter()
            self._listeners: Dict[str, Set[weakref.ref]] =
defaultdict(set)
            self._logger = logging.getLogger(__name__)
            self.initialized = True

    def add_listener(self, target: Any, event_type: str):
        """添加全局事件监听器"""
```

```
        ref = weakref.ref(target, lambda r:
self._cleanup_listener(r, event_type))
        self._listeners[event_type].add(ref)

    def remove_listener(self, target: Any, event_type: str):
        """移除全局事件监听器"""
        to_remove = None
        for ref in self._listeners[event_type]:
            if ref() is target:
                to_remove = ref
                break
        if to_remove:
            self._listeners[event_type].remove(to_remove)

    def _cleanup_listener(self, ref: weakref.ref, event_type:
str):
        """清理失效的监听器"""
        self._listeners[event_type].discard(ref)

    def dispatch_event(self, event: Event, batch: bool = False):
        """分发事件到所有监听器"""
        try:
            if batch:
                self.emitter.add_batch_handler(event.type, lambda
events: self._dispatch_batch(events))
                self.emitter.emit(event)
            else:
                self._dispatch_single(event)
        except Exception as e:
            self._logger.error(f"Error dispatching event: {e}",
exc_info=True)

    def _dispatch_single(self, event: Event):
        """分发单个事件"""
        for ref in list(self._listeners[event.type]):
            target = ref()
            if target is not None:
                try:
```

```
                    target.handle_event(event)
                except Exception as e:
                    self._logger.error(f"Error in event handler:
{e}", exc_info=True)

    def _dispatch_batch(self, events: List[Event]):
        """分发事件批次"""
        if not events:
            return

        event_type = events[0].type
        for ref in list(self._listeners[event_type]):
            target = ref()
            if target is not None:
                try:
                    target.handle_event_batch(events)
                except Exception as e:
                    self._logger.error(f"Error in batch event
handler: {e}", exc_info=True)

    def dispatch_batch(self, events: List[Event]):
        """一次性分发多个事件"""
        for event in events:
            self.dispatch_event(event, batch=True)
```

layouts.py

```
"""
Layout system for PytoWeb
"""

from typing import List, Optional, Union, Tuple, Dict
from .components import Component, Container
from .elements import Element
from .styles import Style

class Grid(Container):
```

```
    """Grid layout component"""

    def __init__(self, columns: int = 12, gap: str = '1rem'):
        super().__init__()
        self.columns = columns
        self.gap = gap

    def render(self) -> Element:
        container = Element('div')
        container.style(
            display='grid',
            grid_template_columns=f'repeat({self.columns}, 1fr)',
            gap=self.gap
        )

        for child in self.children:
            container.add(child.render())

        return container

class Flex(Container):
    """Flexbox layout component"""

    def __init__(self, direction: str = 'row', justify: str =
'flex-start',
                 align: str = 'stretch', wrap: str = 'nowrap'):
        super().__init__()
        self.direction = direction
        self.justify = justify
        self.align = align
        self.wrap = wrap

    def render(self) -> Element:
        container = Element('div')
        container.style(
            display='flex',
            flex_direction=self.direction,
            justify_content=self.justify,
            align_items=self.align,
            flex_wrap=self.wrap
```

```
        )

        for child in self.children:
            container.add(child.render())

        return container

class Responsive(Container):
    """Responsive layout component"""

    def __init__(self, breakpoints: Dict[str, str] = None):
        super().__init__()
        self.breakpoints = breakpoints or {
            'sm': '576px',
            'md': '768px',
            'lg': '992px',
            'xl': '1200px'
        }

    def render(self) -> Element:
        container = Element('div')
        container.style(
            width='100%',
            margin='0 auto',
            padding='0 15px'
        )

        # Add media queries for responsive behavior
        for size, width in self.breakpoints.items():
            container.style(**{
                f'@media (min-width: {width})': {
                    'max-width': width
                }
            })

        for child in self.children:
            container.add(child.render())

        return container
```

router.py

```
"""
PytoWeb 路由模块
"""

from __future__ import annotations
from typing import Dict, List, Callable, Any, Optional, Union,
TypeVar, TYPE_CHECKING
from dataclasses import dataclass
import re
import logging
from http import HTTPStatus

class RouterError(Exception):
    """路由错误"""
    pass

@dataclass
class Route:
    """路由定义类"""
    path: str
    handler: Callable[..., Any]
    methods: List[str]
    name: Optional[str] = None

    def __post_init__(self):
        """验证路由参数"""
        if not self.path.startswith('/'):
            raise RouterError(f"Path must start with '/':
{self.path}")
        if not callable(self.handler):
            raise RouterError(f"Handler must be callable:
{self.handler}")
        if not self.methods:
            self.methods = ['GET']
        self.methods = [m.upper() for m in self.methods]
```

```
        for method in self.methods:
            if method not in ['GET', 'POST', 'PUT', 'DELETE',
'PATCH', 'HEAD', 'OPTIONS']:
                raise RouterError(f"Invalid HTTP method:
{method}")

class Router:
    """路由管理器"""

    def __init__(self):
        self.routes: List[Route] = []
        self._logger = logging.getLogger(__name__)

    def add(self, path: str, handler: Callable[..., Any], methods:
Optional[List[str]] = None, name: Optional[str] = None) -> Router:
        """添加路由"""
        try:
            route = Route(path, handler, methods or ['GET'], name)
            self.routes.append(route)
            return self
        except Exception as e:
            raise RouterError(f"Failed to add route: {e}") from e

    def route(self, path: str, methods: Union[List[str], str] =
'GET', name: Optional[str] = None) -> Callable:
        """通用路由装饰器"""
        if isinstance(methods, str):
            methods = [methods]

        def decorator(handler: Callable[..., Any]) ->
Callable[..., Any]:
            self.add(path, handler, methods, name)
            return handler
        return decorator

    def get(self, path: str, name: Optional[str] = None) ->
Callable:
        """装饰器：添加 GET 路由"""
        return self.route(path, ['GET'], name)
```

```
    def post(self, path: str, name: Optional[str] = None) ->
Callable:
        """装饰器：添加 POST 路由"""
        return self.route(path, ['POST'], name)

    def put(self, path: str, name: Optional[str] = None) ->
Callable:
        """装饰器：添加 PUT 路由"""
        return self.route(path, ['PUT'], name)

    def delete(self, path: str, name: Optional[str] = None) ->
Callable:
        """装饰器：添加 DELETE 路由"""
        return self.route(path, ['DELETE'], name)

    def match(self, path: str, method: str = 'GET') ->
Optional[Route]:
        """匹配路由"""
        method = method.upper()
        try:
            for route in self.routes:
                if method in route.methods:
                    # 简单路径匹配
                    if route.path == path:
                        return route

                    # 参数路径匹配
                    pattern = re.sub(r'{\w+}', r'([^/]+)',
route.path)
                    match = re.match(f'^{pattern}$', path)
                    if match:
                        return route

            return None
        except Exception as e:
            self._logger.error(f"Route matching error: {e}")
            return None
```

```
    def url_for(self, name: str, **params: Any) -> str:
        """根据路由名称生成 URL"""
        try:
            for route in self.routes:
                if route.name == name:
                    url = route.path
                    # 替换 URL 参数
                    for key, value in params.items():
                        placeholder = '{' + key + '}'
                        if placeholder not in url:
                            raise RouterError(f"Parameter '{key}'
not found in route '{name}'")
                        url = url.replace(placeholder, str(value))
                    return url

            raise RouterError(f"No route found with name
'{name}'")
        except Exception as e:
            if isinstance(e, RouterError):
                raise
            raise RouterError(f"Failed to generate URL for route
'{name}': {e}") from e

    def group(self, prefix: str) -> Router:
        """创建路由组"""
        if not prefix.startswith('/'):
            raise RouterError("Group prefix must start with '/'")

        group_router = Router()

        def add_group_route(path: str, handler: Callable[...,
Any], methods: List[str], name: Optional[str] = None) -> None:
            full_path = prefix + path
            self.add(full_path, handler, methods, name)

        group_router.add = add_group_route
        return group_router
```

```
    def mount(self, prefix: str, router: Router) -> Router:
        """挂载其他路由器"""
        if not prefix.startswith('/'):
            raise RouterError("Mount prefix must start with '/'")

        try:
            for route in router.routes:
                full_path = prefix + route.path
                self.add(full_path, route.handler, route.methods,
route.name)
            return self
        except Exception as e:
            raise RouterError(f"Failed to mount router at
'{prefix}': {e}") from e

    def middleware(self, middleware_func: Callable[..., Any]) ->
Router:
        """添加路由中间件"""
        original_routes = self.routes[:]
        for route in original_routes:
            original_handler = route.handler

            def wrapped_handler(*args, **kwargs):
                return middleware_func(original_handler, *args,
**kwargs)

            route.handler = wrapped_handler
        return self

    def dispatch(self, request):
        """Dispatch the request to the appropriate handler"""
        path = request.path
        method = request.method

        # 查找匹配的路由
        handler = self.match(path, method)
        if handler:
            return handler.handler(request)
```

```
        # 没有找到路由
        return None
```

run.py

```
from app import create_app, db

app = create_app()

if __name__ == '__main__':
    with app.app_context():
        db.create_all()
    app.run(debug=True)
```

server.py

```
"""
Server module for PytoWeb
"""

from __future__ import annotations
from typing import Dict, Any, Optional, Callable, List
import logging
import traceback
from http.server import HTTPServer, BaseHTTPRequestHandler
import os
import mimetypes
import json
from urllib.parse import parse_qs, urlparse
import http

class ServerError(Exception):
    """Server error"""
    pass
```

```
class RequestHandler(BaseHTTPRequestHandler):
    """HTTP request handler"""

    def do_GET(self) -> None:
        """Handle GET request"""
        try:
            # Parse URL
            parsed_url = urlparse(self.path)
            path = parsed_url.path
            query = parse_qs(parsed_url.query)

            print(f"[DEBUG] Handling GET request for path:
{path}")
            print(f"[DEBUG] Available routes:
{list(self.server.routes.keys())}")

            # Find route handler
            handler = self.server.routes.get(path)
            if handler:
                try:
                    print(f"[DEBUG] Found handler for path:
{path}")
                    # Call route handler
                    response = handler({"query": query, "method":
"GET"})
                    if response:
                        print(f"[DEBUG] Handler returned response:
{response[:200]}...")
                        self.send_response(http.HTTPStatus.OK)
                        self.send_header('Content-type',
'text/html; charset=utf-8')
                        self.end_headers()
                        self.wfile.write(response.encode('utf-8'))
                    else:
                        print("[DEBUG] Handler returned None")

self.send_error(http.HTTPStatus.INTERNAL_SERVER_ERROR, "Handler
returned None")
                except Exception as e:
                    print(f"[DEBUG] Error in route handler: {e}")
```

```
                    self._log_error(f"Error in route handler:
{e}")

self.send_error(http.HTTPStatus.INTERNAL_SERVER_ERROR)
            else:
                print(f"[DEBUG] No handler found for path:
{path}")
                # Try to serve static file
                static_file = os.path.join(self.server.static_dir,
path.lstrip('/'))
                print(f"[DEBUG] Looking for static file:
{static_file}")
                if os.path.exists(static_file) and
os.path.isfile(static_file):
                    print(f"[DEBUG] Found static file:
{static_file}")
                    self.serve_static_file(static_file)
                else:
                    print(f"[DEBUG] Static file not found:
{static_file}")
                    # Return 404
                    self.send_error(http.HTTPStatus.NOT_FOUND)
        except Exception as e:
            print(f"[DEBUG] Error handling GET request: {e}")
            self._log_error(f"Error handling GET request: {e}")
            self.send_error(http.HTTPStatus.INTERNAL_SERVER_ERROR)

    def do_POST(self) -> None:
        """Handle POST request"""
        try:
            # Read request body
            content_length = int(self.headers.get('Content-
Length', 0))
            post_data =
self.rfile.read(content_length).decode('utf-8')

            try:
                data = json.loads(post_data) if post_data else {}
            except json.JSONDecodeError:
                data = parse_qs(post_data)
```

```
            # Find route handler
            handler = self.server.routes.get(self.path)
            if handler:
                try:
                    # Call route handler
                    response = handler({"data": data, "method":
"POST"})
                    if response:
                        self.send_response(http.HTTPStatus.OK)
                        self.send_header('Content-type',
'text/html; charset=utf-8')
                        self.end_headers()
                        self.wfile.write(response.encode('utf-8'))
                    else:

self.send_error(http.HTTPStatus.INTERNAL_SERVER_ERROR, "Handler
returned None")
                except Exception as e:
                    self._log_error(f"Error in route handler:
{e}")

self.send_error(http.HTTPStatus.INTERNAL_SERVER_ERROR)
            else:
                self.send_error(http.HTTPStatus.NOT_FOUND)
        except Exception as e:
            self._log_error(f"Error handling POST request: {e}")
            self.send_error(http.HTTPStatus.INTERNAL_SERVER_ERROR)

    def serve_static_file(self, filepath: str) -> None:
        """Serve static file"""
        try:
            # Get file MIME type
            content_type = self.guess_type(filepath)

            # Read file content
            with open(filepath, 'rb') as f:
                content = f.read()

            # Send response
```

```
            self.send_response(http.HTTPStatus.OK)
            self.send_header('Content-type', content_type)
            self.send_header('Content-length', str(len(content)))
            self.end_headers()
            self.wfile.write(content)
        except Exception as e:
            self._log_error(f"Error serving static file: {e}")
            self.send_error(http.HTTPStatus.INTERNAL_SERVER_ERROR)

    def guess_type(self, filepath: str) -> str:
        """Guess file MIME type"""
        content_type, _ = mimetypes.guess_type(filepath)
        return content_type or 'application/octet-stream'

    def _log_error(self, message: str) -> None:
        """Log error"""
        logging.error(f"{message}\n{traceback.format_exc()}")

class Server(HTTPServer):
    """PytoWeb server class"""

    def __init__(self, host: str = "localhost", port: int = 8000):
        """Initialize server"""
        super().__init__((host, port), RequestHandler)
        self.routes: Dict[str, Callable] = {}
        self.middleware: List[Callable] = []
        self.static_dir = os.path.join(os.getcwd(), "static")
        self._logger = logging.getLogger(__name__)

        # Create static file directory
        if not os.path.exists(self.static_dir):
            os.makedirs(self.static_dir)

    def add_route(self, path: str, handler: Callable[..., Any]) ->
None:
        """Add route handler"""
        print(f"[DEBUG] Adding route: {path}")
        if not path.startswith('/'):
            path = '/' + path
        self.routes[path] = handler
```

```
        print(f"[DEBUG] Current routes:
{list(self.routes.keys())}")

    def use(self, middleware: Callable[..., Any]) -> None:
        """Add middleware"""
        self.middleware.append(middleware)

    def run(self, host: str, port: int) -> None:
        """Run HTTP server"""
        try:
            print(f"[DEBUG] Starting server at
http://{host}:{port}")
            print(f"[DEBUG] Available routes:
{list(self.routes.keys())}")
            self.serve_forever()
        except Exception as e:
            raise ServerError(f"Failed to start server: {e}")
from e

    def handle_error(self, request: Any, client_address: Any) ->
None:
        """Handle request error"""
        self._logger.error(f"Error handling request from
{client_address}:\n{traceback.format_exc()}")
```

state.py

```
"""State management system for PytoWeb."""
from typing import Dict, List, Any, Callable
import json
import threading
import time
from dataclasses import dataclass
from collections import defaultdict

@dataclass
class StateChange:
    """Represents a state change event."""
```

```
    path: str
    old_value: Any
    new_value: Any

class Store:
    """Central state store with reactive updates."""

    def __init__(self):
        self._state = {}
        self._subscribers = defaultdict(list)
        self._lock = threading.Lock()

    def get(self, path: str, default: Any = None) -> Any:
        """Get state value at path."""
        try:
            keys = path.split('.')
            value = self._state
            for key in keys:
                value = value[key]
            return value
        except (KeyError, TypeError):
            return default

    def set(self, path: str, value: Any):
        """Set state value at path."""
        with self._lock:
            keys = path.split('.')
            target = self._state

            # Navigate to the parent of the target
            for key in keys[:-1]:
                if key not in target:
                    target[key] = {}
                target = target[key]

            # Get old value and set new value
            old_value = target.get(keys[-1])
            target[keys[-1]] = value

            # Notify subscribers
```

```
            change = StateChange(path, old_value, value)
            self._notify_subscribers(change)

    def subscribe(self, path: str, callback:
Callable[[StateChange], None]):
        """Subscribe to state changes at path."""
        with self._lock:
            self._subscribers[path].append(callback)

    def unsubscribe(self, path: str, callback:
Callable[[StateChange], None]):
        """Unsubscribe from state changes at path."""
        with self._lock:
            if path in self._subscribers:
                self._subscribers[path].remove(callback)

    def _notify_subscribers(self, change: StateChange):
        """Notify all relevant subscribers of state change."""
        # Notify exact path subscribers
        for callback in self._subscribers[change.path]:
            callback(change)

        # Notify parent path subscribers
        parts = change.path.split('.')
        for i in range(len(parts)):
            parent_path = '.'.join(parts[:i])
            if parent_path:
                for callback in self._subscribers[parent_path]:
                    callback(change)

class PersistentStore(Store):
    """Store with persistence capabilities."""

    def __init__(self, storage_path: str):
        super().__init__()
        self.storage_path = storage_path
        self._load_state()

    def _load_state(self):
        """Load state from storage."""
```

```
        try:
            with open(self.storage_path, 'r') as f:
                self._state = json.load(f)
        except (FileNotFoundError, json.JSONDecodeError):
            self._state = {}

    def _save_state(self):
        """Save state to storage."""
        with open(self.storage_path, 'w') as f:
            json.dump(self._state, f)

    def set(self, path: str, value: Any):
        """Set state value and persist to storage."""
        super().set(path, value)
        self._save_state()

class StateManager:
    """Manages application state and provides reactive
updates."""

    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance

    def __init__(self, ttl=3600):  # 默认 1 小时过期
        if not hasattr(self, 'initialized'):
            self.store = Store()
            self._state = {}
            self._listeners = {}
            self._batch_updates = False
            self._pending_updates = {}
            self._ttl = ttl
            self._timestamps = {}
            self.initialized = True

    def create_persistent_store(self, name: str, storage_path:
```

```
str) -> PersistentStore:
        """Create a new persistent store."""
        store = PersistentStore(storage_path)
        setattr(self, f"{name}_store", store)
        return store

    def watch(self, paths: List[str], callback:
Callable[[StateChange], None]):
        """Watch multiple paths for changes."""
        for path in paths:
            self.store.subscribe(path, callback)

    def unwatch(self, paths: List[str], callback:
Callable[[StateChange], None]):
        """Unwatch multiple paths."""
        for path in paths:
            self.store.unsubscribe(path, callback)

    def set(self, key: str, value: Any):
        if self._batch_updates:
            self._pending_updates[key] = value
            return

        if key not in self._state or self._state[key] != value:
            self._state[key] = value
            self._timestamps[key] = time.time()
            self._notify_listeners(key)

    def get(self, key: str) -> Any:
        if key in self._state:
            if time.time() - self._timestamps.get(key, 0) >
self._ttl:
                self._cleanup_key(key)
                return None
            return self._state[key]
        return None

    def _cleanup_key(self, key: str):
        self._state.pop(key, None)
        self._timestamps.pop(key, None)
```

```
        if key in self._listeners:
            del self._listeners[key]

    def cleanup_expired(self):
        current_time = time.time()
        expired_keys = [
            key for key, timestamp in self._timestamps.items()
            if current_time - timestamp > self._ttl
        ]
        for key in expired_keys:
            self._cleanup_key(key)

    @classmethod
    def get_instance(cls):
        """Get singleton instance."""
        return cls()
```

styles.py

```
"""
Modern styling system with advanced features
"""

from __future__ import annotations
from typing import Dict, Any, List, Optional, Union, TypeVar,
TYPE_CHECKING
from dataclasses import dataclass

T = TypeVar('T', bound='Style')

@dataclass
class StyleUnit:
    """CSS unit value"""
    value: Union[int, float]
    unit: str = 'px'

    def __str__(self) -> str:
        return f"{self.value}{self.unit}"
```

```
class StyleError(Exception):
    """Style system error"""
    pass

class Style:
    """CSS style management class"""

    def __init__(self, **styles: Any):
        self.rules: Dict[str, str] = {}
        self.add(**styles)

    def add(self: T, **styles: Any) -> T:
        """Add CSS styles"""
        try:
            for key, value in styles.items():
                # Convert Python style names to CSS (e.g.,
font_size -> font-size)
                css_key = key.replace('_', '-')

                # Handle StyleUnit objects
                if isinstance(value, StyleUnit):
                    value = str(value)
                # Handle color tuples (RGB or RGBA)
                elif isinstance(value, tuple):
                    if len(value) == 3:
                        value = f"rgb({value[0]}, {value[1]},
{value[2]})"
                    elif len(value) == 4:
                        value = f"rgba({value[0]}, {value[1]},
{value[2]}, {value[3]})"
                    else:
                        raise StyleError(f"Invalid color tuple
length: {len(value)}")
                # Handle lists (e.g., for multiple background
images)
                elif isinstance(value, list):
                    value = ', '.join(str(v) for v in value)

                self.rules[css_key] = str(value)
```

```
            return self
        except Exception as e:
            raise StyleError(f"Failed to add styles: {e}") from e

    def remove(self: T, *keys: str) -> T:
        """Remove CSS styles"""
        for key in keys:
            css_key = key.replace('_', '-')
            self.rules.pop(css_key, None)
        return self

    def get(self, key: str) -> str:
        """Get style value"""
        css_key = key.replace('_', '-')
        return self.rules.get(css_key, '')

    def to_dict(self) -> Dict[str, str]:
        """Convert to dictionary"""
        return self.rules.copy()

    def to_string(self) -> str:
        """Convert to CSS string"""
        try:
            return '; '.join(f'{k}: {v}' for k, v in
self.rules.items())
        except Exception as e:
            raise StyleError(f"Failed to convert style to string:
{e}") from e

    def to_class_string(self) -> str:
        """Convert to CSS class definition"""
        return ' '.join(self.rules.keys())

    def inline(self) -> str:
        """Convert to inline style string"""
        return self.to_string()

    def update(self: T, **styles: Any) -> T:
        """Update CSS styles"""
        return self.add(**styles)
```

```
    def merge(self: T, other: Style) -> T:
        """Merge with another style"""
        if not isinstance(other, Style):
            raise TypeError("Can only merge with another Style
object")
        new_style = self.__class__()
        new_style.rules.update(self.rules)
        new_style.rules.update(other.rules)
        return new_style

    def clone(self: T) -> T:
        """Create a copy of this style"""
        new_style = self.__class__()
        new_style.rules.update(self.rules)
        return new_style

    def __getattr__(self, name: str) -> str:
        """Get style value using attribute access"""
        return self.get(name)

    def __add__(self: T, other: Style) -> T:
        """Combine two styles"""
        return self.merge(other)

    def __str__(self) -> str:
        """Convert to string"""
        return self.to_string()

class StyleSystem:
    """Modern styling system with advanced features"""

    @staticmethod
    def create_gradient(start_color: str, end_color: str,
direction: str = "to right") -> str:
        """Create linear gradient"""
        return f"linear-gradient({direction}, {start_color},
{end_color})"

    @staticmethod
```

```
    def create_glass_effect(opacity: float = 0.1) -> Dict[str,
str]:
        """Create glass morphism effect"""
        return {
            "background": f"rgba(255, 255, 255, {opacity})",
            "backdrop_filter": "blur(10px)",
            "border": "1px solid rgba(255, 255, 255, 0.2)",
            "box_shadow": "0 8px 32px 0 rgba(31, 38, 135, 0.37)"
        }

    @staticmethod
    def create_neumorphism(color: str, type: str = "flat") ->
Dict[str, str]:
        """Create neumorphism effect"""
        if type == "pressed":
            return {
                "background": color,
                "box_shadow": f"inset 5px 5px 10px rgba(0, 0, 0,
0.1), inset -5px -5px 10px rgba(255, 255, 255, 0.1)"
            }
        else:
            return {
                "background": color,
                "box_shadow": "5px 5px 10px rgba(0, 0, 0, 0.1), -
5px -5px 10px rgba(255, 255, 255, 0.1)"
            }

    @staticmethod
    def create_text_gradient(start_color: str, end_color: str) ->
Dict[str, str]:
        """Create text gradient effect"""
        return {
            "background": f"linear-gradient(to right,
{start_color}, {end_color})",
            "background_clip": "text",
            "text_fill_color": "transparent",
            "-webkit-background-clip": "text",
            "-webkit-text-fill-color": "transparent"
        }
```

```
    @staticmethod
    def create_animation(keyframes: Dict[str, Dict[str, str]],
duration: str = "0.3s", timing: str = "ease") -> Dict[str, str]:
        """Create CSS animation"""
        animation_name = f"animation_{hash(str(keyframes))}"
        keyframe_rules = []

        for selector, styles in keyframes.items():
            style_rules = [f"{k}: {v}" for k, v in styles.items()]
            keyframe_rules.append(f"{selector} {{ {';
'.join(style_rules)} }}")

        keyframe_css = f"@keyframes {animation_name} {{ {'
'.join(keyframe_rules)} }}"

        # TODO: Add keyframe CSS to global styles

        return {
            "animation": f"{animation_name} {duration} {timing}"
        }

    @staticmethod
    def create_transition(properties: List[str], duration: str =
"0.3s", timing: str = "ease") -> str:
        """Create CSS transition"""
        return ", ".join([f"{prop} {duration} {timing}" for prop
in properties])

    @staticmethod
    def create_media_query(breakpoint: str, styles: Dict[str,
str]) -> str:
        """Create media query"""
        return f"@media (min-width: {breakpoint}) {{ {';
'.join([f'{k}: {v}' for k, v in styles.items()])} }}"

    @staticmethod
    def create_hover_effect(styles: Dict[str, str]) -> Dict[str,
str]:
        """Create hover effect styles"""
        return {f"&:hover": styles}
```

```
    @staticmethod
    def create_focus_effect(styles: Dict[str, str]) -> Dict[str,
str]:
        """Create focus effect styles"""
        return {f"&:focus": styles}

    @staticmethod
    def create_active_effect(styles: Dict[str, str]) -> Dict[str,
str]:
        """Create active effect styles"""
        return {f"&:active": styles}

class ModernStyle(Style):
    """Enhanced style class with modern features"""

    def add_glass_effect(self, opacity: float = 0.1):
        """Add glass morphism effect"""
        self.add(**StyleSystem.create_glass_effect(opacity))
        return self

    def add_neumorphism(self, color: str, type: str = "flat"):
        """Add neumorphism effect"""
        self.add(**StyleSystem.create_neumorphism(color, type))
        return self

    def add_text_gradient(self, start_color: str, end_color: str):
        """Add text gradient effect"""
        self.add(**StyleSystem.create_text_gradient(start_color,
end_color))
        return self

    def add_animation(self, keyframes: Dict[str, Dict[str, str]],
duration: str = "0.3s", timing: str = "ease"):
        """Add CSS animation"""
        self.add(**StyleSystem.create_animation(keyframes,
duration, timing))
        return self

    def add_transition(self, properties: List[str], duration: str
```

```
= "0.3s", timing: str = "ease"):
        """Add CSS transition"""

self.add(transition=StyleSystem.create_transition(properties,
duration, timing))
        return self

    def add_hover(self, styles: Dict[str, str]):
        """Add hover effect"""
        self.add(**StyleSystem.create_hover_effect(styles))
        return self

    def add_focus(self, styles: Dict[str, str]):
        """Add focus effect"""
        self.add(**StyleSystem.create_focus_effect(styles))
        return self

    def add_active(self, styles: Dict[str, str]):
        """Add active effect"""
        self.add(**StyleSystem.create_active_effect(styles))
        return self

    def add_responsive(self, breakpoint: str, styles: Dict[str,
str]):
        """Add responsive styles"""
        self.add_raw(StyleSystem.create_media_query(breakpoint,
styles))
        return self

class StylePresets:
    """Predefined modern style presets"""

    @staticmethod
    def button(variant: str = "primary", size: str = "md") ->
Dict[str, str]:
        """Button style preset"""
        base_styles = {
            "border": "none",
            "border_radius": "0.375rem",
            "font_weight": "500",
```

```
            "cursor": "pointer",
            "transition": "all 0.2s ease-in-out"
        }

        # Size variants
        sizes = {
            "sm": {"padding": "0.5rem 1rem", "font_size":
"0.875rem"},
            "md": {"padding": "0.75rem 1.5rem", "font_size":
"1rem"},
            "lg": {"padding": "1rem 2rem", "font_size":
"1.125rem"}
        }

        # Color variants
        variants = {
            "primary": {
                "background": "#3b82f6",
                "color": "#ffffff",
                "&:hover": {"background": "#2563eb"},
                "&:active": {"background": "#1d4ed8"}
            },
            "secondary": {
                "background": "#6b7280",
                "color": "#ffffff",
                "&:hover": {"background": "#4b5563"},
                "&:active": {"background": "#374151"}
            },
            "outline": {
                "background": "transparent",
                "border": "2px solid #3b82f6",
                "color": "#3b82f6",
                "&:hover": {"background": "#3b82f6", "color":
"#ffffff"},
                "&:active": {"background": "#2563eb", "color":
"#ffffff"}
            },
            "ghost": {
                "background": "transparent",
                "color": "#3b82f6",
```

```
                "&:hover": {"background": "rgba(59, 130, 246,
0.1)"},
                "&:active": {"background": "rgba(59, 130, 246,
0.2)"}
            }
        }

        return {**base_styles, **sizes[size], **variants[variant]}

    @staticmethod
    def card(elevation: str = "md") -> Dict[str, str]:
        """Card style preset"""
        base_styles = {
            "background": "#ffffff",
            "border_radius": "0.5rem",
            "padding": "1.5rem",
            "transition": "all 0.2s ease-in-out"
        }

        elevations = {
            "sm": {"box_shadow": "0 1px 2px 0 rgba(0, 0, 0,
0.05)"},
            "md": {"box_shadow": "0 4px 6px -1px rgba(0, 0, 0,
0.1)"},
            "lg": {"box_shadow": "0 10px 15px -3px rgba(0, 0, 0,
0.1)"}
        }

        return {**base_styles, **elevations[elevation]}

    @staticmethod
    def input(variant: str = "outline") -> Dict[str, str]:
        """Input style preset"""
        base_styles = {
            "padding": "0.75rem 1rem",
            "font_size": "1rem",
            "border_radius": "0.375rem",
            "transition": "all 0.2s ease-in-out",
            "&:focus": {
                "outline": "none",
```

```
                "ring": "2px",
                "ring_color": "rgba(59, 130, 246, 0.5)"
            }
        }

        variants = {
            "outline": {
                "border": "1px solid #d1d5db",
                "background": "#ffffff",
                "&:hover": {"border_color": "#9ca3af"},
                "&:focus": {"border_color": "#3b82f6"}
            },
            "filled": {
                "border": "1px solid transparent",
                "background": "#f3f4f6",
                "&:hover": {"background": "#e5e7eb"},
                "&:focus": {"background": "#ffffff",
"border_color": "#3b82f6"}
            },
            "flushed": {
                "border": "none",
                "border_bottom": "1px solid #d1d5db",
                "border_radius": "0",
                "&:hover": {"border_bottom_color": "#9ca3af"},
                "&:focus": {"border_bottom_color": "#3b82f6"}
            }
        }

        return {**base_styles, **variants[variant]}

    @staticmethod
    def badge(variant: str = "primary") -> Dict[str, str]:
        """Badge style preset"""
        base_styles = {
            "display": "inline-flex",
            "align_items": "center",
            "padding": "0.25rem 0.75rem",
            "font_size": "0.875rem",
            "font_weight": "500",
            "border_radius": "9999px",
```

```
            "line_height": "1"
        }

        variants = {
            "primary": {
                "background": "#e0f2fe",
                "color": "#0369a1"
            },
            "success": {
                "background": "#dcfce7",
                "color": "#15803d"
            },
            "warning": {
                "background": "#fff3e0",
                "color": "#ef6c00"
            },
            "error": {
                "background": "#fee2e2",
                "color": "#b91c1c"
            }
        }

        return {**base_styles, **variants[variant]}

# Helper functions for creating style units
def px(value: Union[int, float]) -> StyleUnit:
    """Create pixel unit"""
    return StyleUnit(value, 'px')

def em(value: Union[int, float]) -> StyleUnit:
    """Create em unit"""
    return StyleUnit(value, 'em')

def rem(value: Union[int, float]) -> StyleUnit:
    """Create rem unit"""
    return StyleUnit(value, 'rem')

def percent(value: Union[int, float]) -> StyleUnit:
    """Create percentage unit"""
    return StyleUnit(value, '%')
```

```
def vh(value: Union[int, float]) -> StyleUnit:
    """Create viewport height unit"""
    return StyleUnit(value, 'vh')

def vw(value: Union[int, float]) -> StyleUnit:
    """Create viewport width unit"""
    return StyleUnit(value, 'vw')

# Predefined styles
class Styles:
    """Predefined styles collection"""

    @staticmethod
    def flex(direction: str = 'row', justify: str = 'flex-start',
align: str = 'stretch',
             wrap: bool = False) -> Style:
        """Create flex container style"""
        return Style(
            display='flex',
            flex_direction=direction,
            justify_content=justify,
            align_items=align,
            flex_wrap='wrap' if wrap else 'nowrap'
        )

    @staticmethod
    def grid(columns: int = 12, gap: Union[str, StyleUnit] =
px(16)) -> Style:
        """Create grid container style"""
        return Style(
            display='grid',
            grid_template_columns=f'repeat({columns}, 1fr)',
            gap=str(gap)
        )

    @staticmethod
    def card(shadow: bool = True, radius: Union[str, StyleUnit] =
px(4)) -> Style:
        """Create card style"""
```

```
        style = Style(
            padding=px(16),
            border_radius=str(radius),
            background_color='#ffffff'
        )
        if shadow:
            style.add(box_shadow='0 2px 4px rgba(0,0,0,0.1)')
        return style

    @staticmethod
    def button(variant: str = 'primary') -> Style:
        """Create button style"""
        base_style = Style(
            padding=f'{px(8)} {px(16)}',
            border_radius=px(4),
            border='none',
            cursor='pointer',
            font_weight='500',
            transition='all 0.2s ease'
        )

        variants = {
            'primary': Style(
                background_color='#1976d2',
                color='#ffffff',
                hover={'background_color': '#1565c0'}
            ),
            'secondary': Style(
                background_color='#9e9e9e',
                color='#ffffff',
                hover={'background_color': '#757575'}
            ),
            'outlined': Style(
                background_color='transparent',
                color='#1976d2',
                border='1px solid #1976d2',
                hover={'background_color':
'rgba(25,118,210,0.04)'}
            )
        }
```

```
        return base_style + variants.get(variant,
variants['primary'])

# Default styles
DEFAULT_STYLES = """
.pytoweb-container {
    width: 100%;
    margin: 0 auto;
    padding: 0 16px;
    box-sizing: border-box;
}

.pytoweb-row {
    display: flex;
    flex-wrap: wrap;
    margin: 0 -8px;
}

.pytoweb-col {
    padding: 0 8px;
    box-sizing: border-box;
}

.pytoweb-card {
    background: #ffffff;
    border-radius: 4px;
    padding: 16px;
    box-shadow: 0 2px 4px rgba(0,0,0,0.1);
}

.pytoweb-button {
    display: inline-block;
    padding: 8px 16px;
    border: none;
    border-radius: 4px;
    cursor: pointer;
    font-weight: 500;
    text-align: center;
    transition: all 0.2s ease;
```

```
}

.pytoweb-button:hover {
    opacity: 0.9;
}

.pytoweb-input {
    width: 100%;
    padding: 8px;
    border: 1px solid #ddd;
    border-radius: 4px;
    box-sizing: border-box;
}

.pytoweb-input:focus {
    outline: none;
    border-color: #1976d2;
}

.pytoweb-label {
    display: block;
    margin-bottom: 8px;
    font-weight: 500;
}

.pytoweb-select {
    width: 100%;
    padding: 8px;
    border: 1px solid #ddd;
    border-radius: 4px;
    background-color: #ffffff;
    cursor: pointer;
}

.pytoweb-checkbox {
    margin-right: 8px;
}

.pytoweb-radio {
    margin-right: 8px;
```

```
}

.pytoweb-textarea {
    width: 100%;
    padding: 8px;
    border: 1px solid #ddd;
    border-radius: 4px;
    min-height: 100px;
    resize: vertical;
}

.pytoweb-form {
    width: 100%;
}

.pytoweb-form-group {
    margin-bottom: 16px;
}

.pytoweb-alert {
    padding: 12px;
    border-radius: 4px;
    margin-bottom: 16px;
}

.pytoweb-alert-success {
    background-color: #e8f5e9;
    color: #2e7d32;
}

.pytoweb-alert-error {
    background-color: #ffebee;
    color: #c62828;
}

.pytoweb-alert-warning {
    background-color: #fff3e0;
    color: #ef6c00;
}
```

```
.pytoweb-alert-info {
    background-color: #e3f2fd;
    color: #1565c0;
}
"""
```

themes.py

```
"""
Modern theme system with design tokens and variants
"""

from typing import Dict, Any

class Theme:
    """Theme management class"""

    def __init__(self, name: str = "default"):
        self.name = name
        self.tokens = {
            # Color System
            "colors": {
                # Brand Colors
                "primary": {
                    "main": "#1976d2",
                    "light": "#42a5f5",
                    "dark": "#1565c0",
                    "contrast": "#ffffff"
                },
                "secondary": {
                    "main": "#9c27b0",
                    "light": "#ba68c8",
                    "dark": "#7b1fa2",
                    "contrast": "#ffffff"
                },
                # Semantic Colors
                "success": {
                    "main": "#2e7d32",
```

```
        "light": "#4caf50",
        "dark": "#1b5e20",
        "contrast": "#ffffff"
    },
    "warning": {
        "main": "#ed6c02",
        "light": "#ff9800",
        "dark": "#e65100",
        "contrast": "#ffffff"
    },
    "error": {
        "main": "#d32f2f",
        "light": "#ef5350",
        "dark": "#c62828",
        "contrast": "#ffffff"
    },
    "info": {
        "main": "#0288d1",
        "light": "#03a9f4",
        "dark": "#01579b",
        "contrast": "#ffffff"
    },
    # Gray Scale
    "gray": {
        "50": "#fafafa",
        "100": "#f5f5f5",
        "200": "#eeeeee",
        "300": "#e0e0e0",
        "400": "#bdbdbd",
        "500": "#9e9e9e",
        "600": "#757575",
        "700": "#616161",
        "800": "#424242",
        "900": "#212121"
    },
    # Background & Surface
    "background": {
        "default": "#ffffff",
        "paper": "#ffffff",
        "alt": "#f8f9fa"
```

```
                },
                # Text Colors
                "text": {
                    "primary": "rgba(0, 0, 0, 0.87)",
                    "secondary": "rgba(0, 0, 0, 0.6)",
                    "disabled": "rgba(0, 0, 0, 0.38)",
                    "hint": "rgba(0, 0, 0, 0.38)"
                }
            },

            # Typography System
            "typography": {
                "fontFamily": {
                    "primary": "'Inter', -apple-system,
BlinkMacSystemFont, 'Segoe UI', Roboto, 'Helvetica Neue', Arial,
sans-serif",
                    "code": "'Fira Code', 'Consolas', 'Monaco',
'Andale Mono', monospace"
                },
                "fontWeight": {
                    "light": 300,
                    "regular": 400,
                    "medium": 500,
                    "semibold": 600,
                    "bold": 700
                },
                "fontSize": {
                    "xs": "0.75rem",
                    "sm": "0.875rem",
                    "base": "1rem",
                    "lg": "1.125rem",
                    "xl": "1.25rem",
                    "2xl": "1.5rem",
                    "3xl": "1.875rem",
                    "4xl": "2.25rem",
                    "5xl": "3rem"
                },
                "lineHeight": {
                    "none": 1,
                    "tight": 1.25,
```

```
        "snug": 1.375,
        "normal": 1.5,
        "relaxed": 1.625,
        "loose": 2
    },
    "letterSpacing": {
        "tighter": "-0.05em",
        "tight": "-0.025em",
        "normal": "0",
        "wide": "0.025em",
        "wider": "0.05em",
        "widest": "0.1em"
    }
},

# Spacing System
"spacing": {
    "0": "0",
    "1": "0.25rem",
    "2": "0.5rem",
    "3": "0.75rem",
    "4": "1rem",
    "5": "1.25rem",
    "6": "1.5rem",
    "8": "2rem",
    "10": "2.5rem",
    "12": "3rem",
    "16": "4rem",
    "20": "5rem",
    "24": "6rem",
    "32": "8rem"
},

# Border System
"borders": {
    "width": {
        "none": "0",
        "thin": "1px",
        "medium": "2px",
        "thick": "4px"
```

```
            },
            "radius": {
                "none": "0",
                "sm": "0.125rem",
                "base": "0.25rem",
                "md": "0.375rem",
                "lg": "0.5rem",
                "xl": "0.75rem",
                "2xl": "1rem",
                "full": "9999px"
            },
            "style": {
                "solid": "solid",
                "dashed": "dashed",
                "dotted": "dotted"
            }
        },

        # Shadow System
        "shadows": {
            "none": "none",
            "sm": "0 1px 2px 0 rgba(0, 0, 0, 0.05)",
            "base": "0 1px 3px 0 rgba(0, 0, 0, 0.1), 0 1px
2px 0 rgba(0, 0, 0, 0.06)",
            "md": "0 4px 6px -1px rgba(0, 0, 0, 0.1), 0 2px
4px -1px rgba(0, 0, 0, 0.06)",
            "lg": "0 10px 15px -3px rgba(0, 0, 0, 0.1), 0 4px
6px -2px rgba(0, 0, 0, 0.05)",
            "xl": "0 20px 25px -5px rgba(0, 0, 0, 0.1), 0
10px 10px -5px rgba(0, 0, 0, 0.04)",
            "2xl": "0 25px 50px -12px rgba(0, 0, 0, 0.25)",
            "inner": "inset 0 2px 4px 0 rgba(0, 0, 0, 0.06)"
        },

        # Animation System
        "animation": {
            "duration": {
                "fastest": "75ms",
                "faster": "100ms",
                "fast": "150ms",
```

```
        "normal": "200ms",
        "slow": "300ms",
        "slower": "400ms",
        "slowest": "500ms"
    },
    "easing": {
        "linear": "linear",
        "ease": "ease",
        "easeIn": "cubic-bezier(0.4, 0, 1, 1)",
        "easeOut": "cubic-bezier(0, 0, 0.2, 1)",
        "easeInOut": "cubic-bezier(0.4, 0, 0.2, 1)"
    }
},

# Z-index System
"zIndex": {
    "hide": -1,
    "base": 0,
    "dropdown": 1000,
    "sticky": 1100,
    "fixed": 1200,
    "modalBackdrop": 1300,
    "modal": 1400,
    "popover": 1500,
    "tooltip": 1600
},

# Breakpoints System
"breakpoints": {
    "xs": "0px",
    "sm": "600px",
    "md": "900px",
    "lg": "1200px",
    "xl": "1536px"
},

# Grid System
"grid": {
    "columns": 12,
    "gutter": {
```

```
                    "xs": "1rem",
                    "sm": "1.5rem",
                    "md": "2rem",
                    "lg": "2.5rem",
                    "xl": "3rem"
                },
                "margin": {
                    "xs": "1rem",
                    "sm": "1.5rem",
                    "md": "2rem",
                    "lg": "2.5rem",
                    "xl": "3rem"
                }
            }
        }

    def get_token(self, path: str) -> Any:
        """Get design token value by path"""
        keys = path.split(".")
        value = self.tokens
        for key in keys:
            value = value.get(key)
            if value is None:
                return None
        return value

    def set_token(self, path: str, value: Any):
        """Set design token value by path"""
        keys = path.split(".")
        target = self.tokens
        for key in keys[:-1]:
            target = target.setdefault(key, {})
        target[keys[-1]] = value

    def create_variant(self, name: str, overrides: Dict[str, Any])
-> "Theme":
        """Create theme variant with overrides"""
        variant = Theme(f"{self.name}-{name}")
        variant.tokens = self.tokens.copy()
```

```
    for path, value in overrides.items():
        variant.set_token(path, value)

    return variant

@classmethod
def create_dark_theme(cls) -> "Theme":
    """Create dark theme variant"""
    dark_theme = cls("dark")
    dark_theme.tokens["colors"].update({
        "background": {
            "default": "#121212",
            "paper": "#1e1e1e",
            "alt": "#2c2c2c"
        },
        "text": {
            "primary": "rgba(255, 255, 255, 0.87)",
            "secondary": "rgba(255, 255, 255, 0.6)",
            "disabled": "rgba(255, 255, 255, 0.38)",
            "hint": "rgba(255, 255, 255, 0.38)"
        }
    })
    return dark_theme

@classmethod
def create_high_contrast_theme(cls) -> "Theme":
    """Create high contrast theme variant"""
    high_contrast = cls("high-contrast")
    high_contrast.tokens["colors"].update({
        "background": {
            "default": "#000000",
            "paper": "#000000",
            "alt": "#1a1a1a"
        },
        "text": {
            "primary": "#ffffff",
            "secondary": "#ffffff",
            "disabled": "#808080",
            "hint": "#808080"
        }
```

```
        })
        return high_contrast

class ThemeProvider:
    """Theme provider for components"""

    _current_theme: Theme = None

    @classmethod
    def set_theme(cls, theme: Theme):
        """Set current theme"""
        cls._current_theme = theme

    @classmethod
    def get_theme(cls) -> Theme:
        """Get current theme"""
        return cls._current_theme

    @classmethod
    def use_theme(cls) -> Theme:
        """Get current theme or default"""
        if cls._current_theme is None:
            cls._current_theme = Theme({})
        return cls._current_theme
```

validation.py

```
"""Form validation system for PytoWeb"""
from typing import Dict, Any, List, Optional, Callable
import re

class ValidationRule:
    """Base class for validation rules"""
    def __init__(self, message: str):
        self.message = message

    def validate(self, value: Any) -> bool:
        raise NotImplementedError
```

```
class Required(ValidationRule):
    """Required field validation"""
    def __init__(self, message: str = "This field is required"):
        super().__init__(message)

    def validate(self, value: Any) -> bool:
        if value is None:
            return False
        if isinstance(value, str) and not value.strip():
            return False
        return True

class MinLength(ValidationRule):
    """Minimum length validation"""
    def __init__(self, min_length: int, message: str = None):
        super().__init__(message or f"Minimum length is
{min_length}")
        self.min_length = min_length

    def validate(self, value: str) -> bool:
        return len(str(value)) >= self.min_length

class MaxLength(ValidationRule):
    """Maximum length validation"""
    def __init__(self, max_length: int, message: str = None):
        super().__init__(message or f"Maximum length is
{max_length}")
        self.max_length = max_length

    def validate(self, value: str) -> bool:
        return len(str(value)) <= self.max_length

class Pattern(ValidationRule):
    """Pattern validation using regex"""
    def __init__(self, pattern: str, message: str = "Invalid
format"):
        super().__init__(message)
        self.pattern = re.compile(pattern)
```

```
    def validate(self, value: str) -> bool:
        return bool(self.pattern.match(str(value)))

class Email(ValidationRule):
    """Email format validation"""
    def __init__(self, message: str = "Invalid email format"):
        super().__init__(message)
        self.pattern = re.compile(r"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-
9.-]+\.[a-zA-Z]{2,}$")

    def validate(self, value: str) -> bool:
        return bool(self.pattern.match(str(value)))

class Range(ValidationRule):
    """Numeric range validation"""
    def __init__(self, min_value: float = None, max_value: float
= None, message: str = None):
        message = message or f"Value must be between {min_value}
and {max_value}"
        super().__init__(message)
        self.min_value = min_value
        self.max_value = max_value

    def validate(self, value: float) -> bool:
        try:
            num = float(value)
            if self.min_value is not None and num <
self.min_value:
                return False
            if self.max_value is not None and num >
self.max_value:
                return False
            return True
        except (TypeError, ValueError):
            return False

class Custom(ValidationRule):
    """Custom validation using a callback function"""
    def __init__(self, validator: Callable[[Any], bool], message:
str):
```

```
        super().__init__(message)
        self.validator = validator

    def validate(self, value: Any) -> bool:
        return self.validator(value)

class FormValidator:
    """Form validation manager"""
    def __init__(self):
        self.fields: Dict[str, List[ValidationRule]] = {}
        self.errors: Dict[str, List[str]] = {}

    def add_field(self, field_name: str, rules:
List[ValidationRule]):
        """Add validation rules for a field"""
        self.fields[field_name] = rules

    def validate(self, data: Dict[str, Any]) -> bool:
        """Validate form data"""
        self.errors.clear()
        is_valid = True

        for field_name, rules in self.fields.items():
            field_value = data.get(field_name)
            field_errors = []

            for rule in rules:
                if not rule.validate(field_value):
                    field_errors.append(rule.message)
                    is_valid = False

            if field_errors:
                self.errors[field_name] = field_errors

        return is_valid

    def get_errors(self) -> Dict[str, List[str]]:
        """Get validation errors"""
        return self.errors
```

vdom.py

```
"""Virtual DOM implementation for PytoWeb."""
from typing import Dict, List, Optional, Any
import difflib

class VNode:
    """Virtual DOM Node."""
    def __init__(self, tag: str, props: Dict = None, children:
List = None):
        self.tag = tag
        self.props = props or {}
        self.children = children or []
        self.key = props.get('key') if props else None

    def __eq__(self, other):
        if not isinstance(other, VNode):
            return False
        return (self.tag == other.tag and
                self.props == other.props and
                self.children == other.children)

class VDOMDiffer:
    """Handles virtual DOM diffing and patching."""

    @staticmethod
    def diff(old_node: Optional[VNode], new_node: Optional[VNode])
-> List[Dict]:
        """Generate a list of patches based on differences
between nodes."""
        patches = []

        if old_node is None:
            patches.append({
                'type': 'CREATE',
                'node': new_node
            })
        elif new_node is None:
```

```
            patches.append({
                'type': 'REMOVE'
            })
        elif old_node != new_node:
            if old_node.tag != new_node.tag:
                patches.append({
                    'type': 'REPLACE',
                    'node': new_node
                })
            else:
                # Props diff
                props_patch =
VDOMDiffer._diff_props(old_node.props, new_node.props)
                if props_patch:
                    patches.append({
                        'type': 'PROPS',
                        'props': props_patch
                    })

                # Children diff
                children_patches = VDOMDiffer._diff_children(
                    old_node.children,
                    new_node.children
                )
                patches.extend(children_patches)

        return patches

    @staticmethod
    def _diff_props(old_props: Dict, new_props: Dict) ->
Optional[Dict]:
        """Compare props and return differences."""
        props_patch = {}

        # Check for changed or new props
        for key, value in new_props.items():
            if key not in old_props or old_props[key] != value:
                props_patch[key] = value

        # Check for removed props
```

```
        for key in old_props:
            if key not in new_props:
                props_patch[key] = None

        return props_patch if props_patch else None

    @staticmethod
    def _diff_children(old_children: List[VNode], new_children:
List[VNode]) -> List[Dict]:
        """Compare children nodes and return patches."""
        patches = []

        # Use difflib for optimal diff
        matcher = difflib.SequenceMatcher(None, old_children,
new_children)
        for tag, i1, i2, j1, j2 in matcher.get_opcodes():
            if tag == 'replace':
                for i in range(i1, i2):
                    patches.append({
                        'type': 'REPLACE_CHILD',
                        'index': i,
                        'node': new_children[j1 + (i - i1)] if i
- i1 < j2 - j1 else None
                    })
            elif tag == 'delete':
                for i in range(i1, i2):
                    patches.append({
                        'type': 'REMOVE_CHILD',
                        'index': i
                    })
            elif tag == 'insert':
                for j in range(j1, j2):
                    patches.append({
                        'type': 'INSERT_CHILD',
                        'index': j,
                        'node': new_children[j]
                    })

        return patches
```

```
class VDOMRenderer:
    """Handles rendering virtual DOM to real DOM."""
    _string_pool = {}
    _pool_size = 1000

    @staticmethod
    def _get_pooled_string(s: str) -> str:
        if s not in VDOMRenderer._string_pool:
            if len(VDOMRenderer._string_pool) >=
VDOMRenderer._pool_size:
                VDOMRenderer._string_pool.clear()
            VDOMRenderer._string_pool[s] = s
        return VDOMRenderer._string_pool[s]

    @staticmethod
    def create_element(vnode: VNode) -> str:
        if isinstance(vnode, str):
            return VDOMRenderer._get_pooled_string(vnode)

        void_elements = {
            'area', 'base', 'br', 'col', 'embed', 'hr', 'img',
'input',
            'link', 'meta', 'param', 'source', 'track', 'wbr'
        }

        tag = VDOMRenderer._get_pooled_string(vnode.tag)
        html = ['<{}'.format(tag)]

        if vnode.props:
            props_str = VDOMRenderer._props_to_string(vnode.props)

html.append(VDOMRenderer._get_pooled_string(props_str))

        if tag in void_elements:
            html.append('/>')
            return VDOMRenderer._get_pooled_string(''.join(html))

        html.append('>')

        if vnode.children:
```

```
            for child in vnode.children:
                if isinstance(child, VNode):

html.append(VDOMRenderer.create_element(child))
                else:
                    html.append(str(child))

        html.append('</{}>'.format(tag))
        return VDOMRenderer._get_pooled_string(''.join(html))

    @staticmethod
    def _props_to_string(props: Dict) -> str:
        """Convert props dictionary to HTML attributes string."""
        if not props:
            return ''

        attributes = []
        for key, value in props.items():
            if value is None or value is False:
                continue
            if value is True:
                attributes.append(key)
            else:
                # 处理事件处理器
                if key.startswith('on'):
                    # 将 Python 函数转换为 JavaScript 事件处理器
                    value = "pytoWeb.handleEvent('{}',
this)".format(key)
                # 处理样式对象
                elif key == 'style' and isinstance(value, dict):
                    value = ';'.join('{}:{}'.format(k, v) for k,
v in value.items())
                # 处理类名列表
                elif key == 'class' and isinstance(value, (list,
set)):
                    value = ' '.join(value)

                attributes.append('{}="{}"'.format(key,
```

```
str(value).replace('"', "&quot;")))

        return ' ' + ' '.join(attributes) if attributes else ''

    @staticmethod
    def render_to_string(vnode: Any) -> str:
        """Render a virtual DOM node to HTML string."""
        try:
            if isinstance(vnode, (str, int, float)):
                return str(vnode)
            elif isinstance(vnode, VNode):
                return VDOMRenderer.create_element(vnode)
            elif hasattr(vnode, 'render'):
                # Handle components that have a render method
                rendered = vnode.render()
                return VDOMRenderer.render_to_string(rendered)
            elif isinstance(vnode, (list, tuple)):
                # Handle lists of nodes
                return
''.join(VDOMRenderer.render_to_string(child) for child in vnode)
            else:
                return str(vnode)
        except Exception as e:
            raise Exception(f"Failed to render node: {e}")
```

workers.py

```
"""Web Worker support for PytoWeb."""
import json
import threading
import queue
import logging
import traceback
from typing import Dict, Any, Callable, Optional
from dataclasses import dataclass
from concurrent.futures import ThreadPoolExecutor

@dataclass
```

```
class WorkerMessage:
    """Message passed between main thread and worker."""
    type: str
    data: Any
    id: Optional[str] = None

class PythonWorker:
    """Python-based worker implementation."""

    def __init__(self, name: str):
        self.name = name
        self._running = False
        self._thread: Optional[threading.Thread] = None
        self._message_queue = queue.Queue()
        self._callbacks: Dict[str, Callable] = {}
        self._error_handler: Optional[Callable] = None

    def start(self):
        """Start the worker thread."""
        if self._running:
            return

        self._running = True
        self._thread = threading.Thread(target=self._run)
        self._thread.daemon = True
        self._thread.start()

    def stop(self):
        """Stop the worker thread."""
        self._running = False
        if self._thread:
            self._thread.join()

    def post_message(self, message_type: str, data: Any,
message_id: Optional[str] = None):
        """Send a message to the worker."""
        message = WorkerMessage(type=message_type, data=data,
id=message_id)
        self._message_queue.put(message)
```

```
    def on_message(self, message_type: str, callback: Callable):
        """Register a message handler."""
        self._callbacks[message_type] = callback

    def on_error(self, handler: Callable):
        """Register an error handler."""
        self._error_handler = handler

    def _run(self):
        """Main worker loop."""
        while self._running:
            try:
                message = self._message_queue.get(timeout=1.0)
                self._handle_message(message)
            except queue.Empty:
                continue
            except Exception as e:
                if self._error_handler:
                    self._error_handler(e)
                else:
                    logging.error(f"Worker error:
{str(e)}\n{traceback.format_exc()}")

    def _handle_message(self, message: WorkerMessage):
        """Handle a received message."""
        if message.type in self._callbacks:
            try:
                result =
self._callbacks[message.type](message.data)
                if message.id:
                    # 如果消息有 ID，发送响应
                    self.post_message('response', {
                        'id': message.id,
                        'result': result
                    })
            except Exception as e:
                if message.id:
                    # 发送错误响应
                    self.post_message('error', {
```

```
                        'id': message.id,
                        'error': str(e)
                    })
                raise

class WorkerPool:
    """Manages a pool of workers."""

    def __init__(self, size: int = 4):
        self._workers: Dict[str, PythonWorker] = {}
        self._executor = ThreadPoolExecutor(max_workers=size)
        self._size = size

    def create_worker(self, name: str) -> PythonWorker:
        """Create a new worker."""
        if name in self._workers:
            raise ValueError(f"Worker '{name}' already exists")

        worker = PythonWorker(name)
        self._workers[name] = worker
        worker.start()
        return worker

    def get_worker(self, name: str) -> Optional[PythonWorker]:
        """Get an existing worker."""
        return self._workers.get(name)

    def remove_worker(self, name: str):
        """Remove and stop a worker."""
        if name in self._workers:
            worker = self._workers.pop(name)
            worker.stop()

    def stop_all(self):
        """Stop all workers."""
        for worker in self._workers.values():
            worker.stop()
        self._workers.clear()
        self._executor.shutdown()
```

```
class WorkerDecorators:
    """Decorators for worker functionality."""

    @staticmethod
    def run_in_worker(worker_name: str):
        """Decorator to run a function in a worker."""
        def decorator(func):
            def wrapper(*args, **kwargs):
                worker = WorkerPool().get_worker(worker_name)
                if not worker:
                    worker =
WorkerPool().create_worker(worker_name)

                # 创建消息 ID
                message_id =
f"{func.__name__}_{id(args)}_{id(kwargs)}"

                # 创建 Future 对象
                future = threading.Event()
                result = {'value': None, 'error': None}

                def handle_response(data):
                    if data['id'] == message_id:
                        if 'result' in data:
                            result['value'] = data['result']
                        else:
                            result['error'] = data['error']
                        future.set()

                worker.on_message('response', handle_response)
                worker.post_message(func.__name__, {
                    'args': args,
                    'kwargs': kwargs
                }, message_id)

                # 等待结果
                future.wait()
```

```
                if result['error']:
                    raise Exception(f"Error in worker:
{result['error']}")
                return result['value']

            return wrapper
        return decorator

    @staticmethod
    def worker_method(message_type: str):
        """Decorator to register a worker method."""
        def decorator(func):
            def wrapper(self, *args, **kwargs):
                if isinstance(self, PythonWorker):
                    return func(self, *args, **kwargs)
                else:
                    raise TypeError("Decorator must be used with
PythonWorker class")

            # 注册消息处理器
            if hasattr(func, '__self__') and
isinstance(func.__self__, PythonWorker):
                func.__self__.on_message(message_type, wrapper)

            return wrapper
        return decorator
```