# プロジェクトコードネーム: "Ouroboros" (The Snake)

## C++標準準拠・高性能Webサーバーライブラリ 要件・仕様・設計定義書

作成者: シニア・システムソフトウェアエンジニア (AI代理)
対象プラットフォーム: Linux (Kernel 5.10+, io_uring必須)
言語標準: C++20 / C++23

## 1. 要件定義書 (Requirements Definition)

### 1.1 プロジェクトの目的と哲学

既存のフレームワーク（Boost.Asio等）や他言語（C#, Rust, Go）のWebサーバーが持つオーバーヘッドを徹底的に排除し、Linuxカーネルの性能を極限まで引き出すWebサーバーライブラリを構築する。
C++標準ライブラリ（STL）との完全な調和を目指し、スネークケース (snake_case) を採用することで、ユーザーコードにおいてSTLコンテナと本ライブラリの境界を感じさせない「言語拡張」レベルの統合を実現する。

### 1.2 ターゲットユースケース

- **超低レイテンシ・マイクロサービス:** ミリ秒以下の応答速度が求められる金融取引、リアルタイム入札（RTB）システム。
- **エッジコンピューティング:** リソース制限のある環境下での、メモリ割り当てを極小化した高スループット処理。
- **組み込みハイパフォーマンス:** 外部ライブラリ依存（OpenSSL以外）を排除した、軽量かつ堅牢な通信基盤。

### 1.3 機能要件

- **プロトコル:** HTTP/1.1 (Pipelining対応), 将来的なHTTP/2対応を見越した抽象化。
- **アーキテクチャ:** シングルスレッド（イベントループ）および Thread-per-core モデルのサポート。
- **ルーティング:** コンパイル時正規表現、またはトライ木（Trie）ベースの高速パスルーティング。
- **将来の拡張性 (The "www" Vision):**
  - 現在は通信層（http）に注力するが、将来構想として React/Vue ライクなコンポーネント指向Webアプリフレームワークを ouroboros::www 名前空間に実装するための基盤を提供する。
  - 高速なサーバーサイドレンダリング（SSR）と、コンポーネントツリーの効率的な構築を可能にする設計とする。
- **依存関係:**
  - **許可:** Linux System Calls (<sys/socket.h\>, <linux/io_uring.h\>), C++ Standard Library.
  - **禁止:** Boost, Asio, POCO, その他重量級フレームワーク。

### 1.4 パフォーマンス目標 (KPI)

- **スループット:** localhostベンチマークにおいて、ASP.NET Core (Kestrel) および Rust (Actix-web) と同等以上。
- **メモリ効率:** アイドル時のフットプリント最小化。リクエスト処理中のヒープ割り当て回数（new/malloc）を **0回** に近づける。
- **レイテンシ:** テールレイテンシ（p99）の安定化。

## 2. 仕様書 (Specification)

### 2.1 APIデザインと命名規則

STLの哲学に従い、すべての識別子はスネークケースとする。クラス名は名詞、メソッド名は動詞とするが、パスカルケース（MyClass）は一切使用しない。

- **名前空間構成:**

    - ouroboros: ルート名前空間。
    - ouroboros::http: 低レイテンシHTTPサーバー、クライアント、WebSocket、io_uring ラッパー等の通信コア機能。
    - ouroboros::www: (Future) UIコンポーネント、SSRエンジン、ステート管理などのWebアプリケーション層。
- **主要クラス:** unique_socket, io_context, request, response, server (これらは ouroboros::http 内に配置)

**ユーザーコード例**

```
#include "ouroboros/http/server.hpp"

// 将来的なアプリケーション層のイメージ
// #include "ouroboros/www/component.hpp"

int main() {
    using namespace ouroboros;

    http::io\_context ctx;
    http::server server(ctx);

    // ルート登録: STLライクなインターフェース
    server.get("/api/status", \[\](const auto& req) \-\> http::response {
        return {200, "application/json", R"({"status":"ok"})"};
    });

    // パスパラメータのゼロコピー取得
    server.get("/users/:id", \[\](const auto& req) {
        // idは元のバッファへのビュー (std::string\_view)
        auto user\_id \= req.params\["id"\];
        return lookup\_user(user\_id);
    });

    server.listen(8080);
    ctx.run(); // ブロッキング・イベントループ
}
```

### 2.2 ネットワークI/Oモデル

- **非同期基盤:** io_uring を全面的に採用。epoll などのレガシーAPIは使用しない。
- **完了キュー:** io_uring の CQE (Completion Queue Entry) をポーリングし、対応するコールバックまたはコルーチンを再開する。
- **バッファリング:** IO_URING_OP_PROVIDE_BUFFERS を使用し、カーネルが自動的にバッファを選択する仕組みを取り入れ、システムコール発行回数を削減する。

### 2.3 エラーハンドリング

例外（Exceptions）は「回復不可能なエラー（起動時の設定ミス、メモリ枯渇）」にのみ使用する。
ランタイムの制御フロー（接続切断、パースエラー、404等）には std::expected (C++23) またはそれに準ずる result 型を使用する。

```
// 戻り値で成功/失敗を明示
std::expected\<size\_t, std::error\_code\> send\_result \= socket.send(data);
if (\!send\_result) {
    // エラー処理
}
```

## 3. 設計書 (Internal Architecture)

### 3.1 クラス構造: unique_socket を起点として

本ライブラリの「背骨」となるのは、RAIIを徹底したファイル記述子（FD）のラッパーである。

**ouroboros::http::unique_socket (Core)**

`std::unique\_ptr` のような所有権セマンティクスを持つ。

- **責務:**
  - socket FDの保持とライフサイクル管理（デストラクタで close()）。
  - コピー禁止、ムーブのみ許可。
  - 暗黙の型変換（int へのキャスト）は禁止し、.native_handle() で明示的にアクセスさせる。

**ouroboros::http::io_context (Event Loop)**

- **責務:** io_uring インスタンスの初期化と管理。
- **メンバ:**
  - `struct io_uring ring_` : liburingを使わず、生のカーネル構造体をラップする。
  - `submit_request()` : SQE (Submission Queue Entry) を発行する。
  - `run()` : イベントループのメイン。CQEを取り出し、関連付けられた user_data ポインタ（Task構造体）を実行する。

## 3.2 ゼロコピー・アーキテクチャ

リクエスト受信からレスポンス送信まで、データのコピーを徹底的に避ける。

1. **受信バッファ (Ring Buffer):**
   - 固定サイズのメモリプール `std::vector<std::byte>` を事前に確保。
   - `io_uring` にバッファグループとして登録。
2. **HTTPパーサー (State Machine):**
   - 受信したバイト列に対して、ポインタ操作のみで解析を行う。
   - ** `std::string` は生成しない。**
   - request オブジェクトは、内部に受信バッファへの `std::string_view` のみを保持する。
   - ヘッダー名、値、ボディ、すべてが `view` であるため、パースに伴うアロケーションはゼロ。

## 3.3 スレッドモデル (Thread-per-Core)

- **共有なしアーキテクチャ (Shared-nothing):**
  - 各CPUコアに1つのスレッド、1つの `io_context` 、1つの `io_uring` リングを割り当てる。
  - `SO_REUSEPORT` を使用し、すべてのスレッドが同じポート（例: 80）をリッスンする。
  - カーネルが接続を各スレッドにロードバランスするため、ユーザーランドでのロック競合（Mutex）が発生しない。

## 3.4 ライフサイクル管理フロー

1. **初期化:** `server` が `unique_socket` を作成し、`bind/listen` する。
2. **Accept:** `io_uring` に `IORING_OP_ACCEPT` を発行。
3. **接続確立:** 完了時、新しい `unique_socket` (クライアント用) が生成される。
4. **Read:** `IORING_OP_RECV` を発行。バッファはカーネルがプールから選択。
5. **Parse & Handle:** データ到着時、パーサーが起動。完了すればユーザー定義のハンドラ（ラムダ等）を実行。
6. **Write:** `IORING_OP_SEND` を発行。 `std::span` や `iovec` を使用してスキャッターギャザーI/Oを行う。
7. **Close:** 通信終了またはタイムアウト時、`unique_socket` がスコープアウトし、リソースが自動解放される。

## 4. 実装に向けた技術メモ (Technical Notes)

- **C++20 Concepts:** テンプレート引数の制約に `concept` を積極的に使用し、コンパイルエラーの可読性を高める。
  - 例: `template <typename Handler> requires std::invocable<Handler, const request&>`
- **Syscall Wrapper:** 生のシステムコール（ `io_uring_setup` , `io_uring_enter` 等）をラップするインライン関数群を用意し、可読性を確保しつつオーバーヘッドをなくす。

- **安全性:** `reinterpret_cast` は `io_uring` との境界（`user_data` の変換）のみに限定し、それ以外は型安全なC++キャストを使用する。