1対多非同期結合型デバッガによるwebアプリケーションのセッションアウェア追跡

小菅圭介† 佐藤規男†

Ruby および Python は，複雑な分散マルチスレッドや web アプリケーションに適した先進的オブジェクト指向スクリプト言語である．特に，web アプリケーションでの需要は非常に大きい．本論文では，web アプリケーションのためのシンボリックデバッグ環境を提案する．複雑な web アプリケーションでは，HTTP リクエストに含まれる session ID をキーとして，データを共有する多数の web スクリプトが個別にイベント駆動される．よって，web スクリプトを連繋動作させたときのセッションとしての通しデバッグと，複数セッションの相互作用デバッグが必要である．しかし，その実行プラットフォームである web サーバにおいては，スレッドは複数のセッションで使い回され，かつ，セッションは複数のスレッドにより実行されるため，スレッドアウェアなデバッグ機能のみでこの種のデバッグをするには不十分である．そこで，以下のようなセッションアウェアなデバッグ機能を提案する．(1) session ID の自動取得ならびに，session ID によるスレッドのグループ化表示．(2) セッションを指定した形での，コントローラ部のアクション起動時におけるブレークポイントの設定ならびにオン・オフする機能．上記（1）と（2）の機能を，Rails アプリケーションの典型的実行プラットフォームを対象として，既開発の非同期型スレッドアウェアデバッガ（Dionea）をベースに実装した．

Session-aware Trace of Web Applications by One-to-Many Asynchronously Coupled Debugger

Keisuke Kosuga† and Norio Sato†

Both Ruby and Python are innovative scripting languages that are suitable for complex distributed multi-thread programming and web applications, of which the demand for the latter is huge. This paper proposes a symbolic debugging environment specialized for web applications. Complex web applications are executed in an event-driven way by individual web scripts that share the same data whose key is the “session ID” in HTTP requests coming from a browser. Therefore, the debugging of their session-through behaviors and the intractions among sessions are needed. Since in their execution platforms, whch are web servers, one thread is reused for more than one sessions and one session involves more than one threads, thread-aware features alone do not suffice for such a kind of debugging. We propose, therefore, the following “session-aware” features: (1) Acquiring “session ID” automatically, and showing threads grouped by “session ID”. (2) For a specified session, setting a breakpoint at an action of controller for each its invocation, and enabling/disabling this feature. We have implemented features (1) and (2), for several representative web application platforms that are used for Rails applications, based upon our previous work of asynchronous type thread-aware debugger named “Dionea”.

1. はじめに

Python[1] と Ruby[2] は，マルチスレッド記述の言語構造とライブラリ，さらに，dRuby[3),4)] など分散処理プラットフォームを持ち，さらに最近では web アプリケーションを作成するためのフレームワーク☆Rails[9)] がオープンソースとして提供されるなど，複雑なネットワークプログラムを実装するための生産環境が充実している．特に，Ruby による web アプリケーションの生産性は，Rails フレームワークの実用化により加速され，続いて Python 用フレームワーク Django☆☆や TurboGears☆☆☆も実用化への準備が進みつつある．web アプリケーションの需要にもかかわらず，テスト環境の方

† 金沢工業大学大学院情報工学専攻
Department of Information and Computer Science, Kanazawa Institute of Technology

☆ 広範なライブラリ，スケルトンやテンプレートを生成するためのツールなどからなる．
☆☆ http://www.djangoproject.com
☆☆☆ http://www.turbogears.org

は，主に web サーバのログ解析に頼るのが現状である．ごく最近になって，ブラウザシミュレータ Selenium-IDE☆がオープン化され，ユニットテスト[12] とあわせて，black-box テスト環境というアプローチでの関心は高まってきている．

本論文では，white-box テスト環境の典型である対話型シンボリックデバッグというアプローチからの提案を行う．特に，“Ruby on Rails” による複雑な web アプリケーションを対象とした提案である．実装は，既開発の非同期型スレッドアウェアデバッガ Dionea☆☆,[7),8)] をベースとする．また，適用実験例に基づき適用シナリオおよび評価を述べる．

以降，2 章では，目標である web アプリケーション向けデバッグ環境の提案を行う．3 章では，実装のベースとなる既存 Dionea の機能と実装をまとめる．4 章では，提案機能の実装を述べる．5 章では，適用シナリオを述べ，6 章では，考察と評価を述べる．7 章では，今後の課題を述べる．

2. web アプリケーション向けデバッグ機能の提案

2.1 web アプリケーションの特徴

HTTP はステートレスなプロトコルであるため，web アプリケーションは，1 つまたはそれ以上のプログラム部分（以降 web スクリプトと呼ぶ☆☆☆）から構成される．**図 1** に示すように，web スクリプトはリクエストごとに駆動されるが，典型的には，ブラウザ内部で保持される cookie が表す session ID をキーとするセッションデータを，サーバ側でを引き継ぐことにより連繋動作する．この一連の実行をセッションと呼ぶ．他方，web サーバ内部においてスレッドがどのように実行されるかは，タイミングと web サーバの構造に依存する．**図 2** に示すように，たとえば，HTTP 1.1 の “keep-alive” のタイムアウトにより，1 つのスレッドが異なるセッションの異なる web スクリプトを実行することがある．あるいは，同一のスレッドが同一のセッションを実行すること，異なるスレッドが同一セッションを実行すること，などが起こりうる．

開発者は，個別のスクリプトを実装しながらツールなしで全体の挙動を頭に描かなければならない．リクエストによるスクリプトのイベント駆動は，ブラウザ操作特有の back page アクセスも想定すれば，その組合せが多く，また，セッション情報の共有のみによるスクリプトの連繋は，1 つのスレッドにより状態変化を保持することに比べて，想定外のリクエストに対するガードがゆるく，これは，開発者に多大な労力を要求することになる．

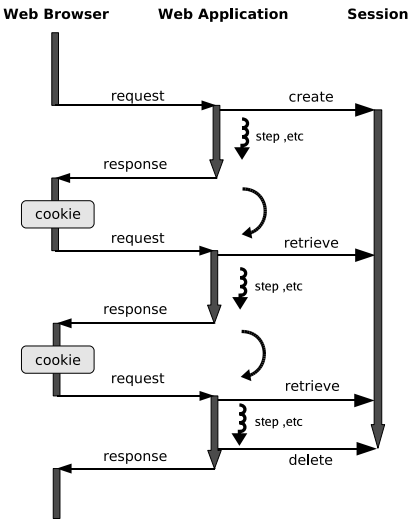

図 1 スレッドとセッション
Fig. 1 Threads and a session.

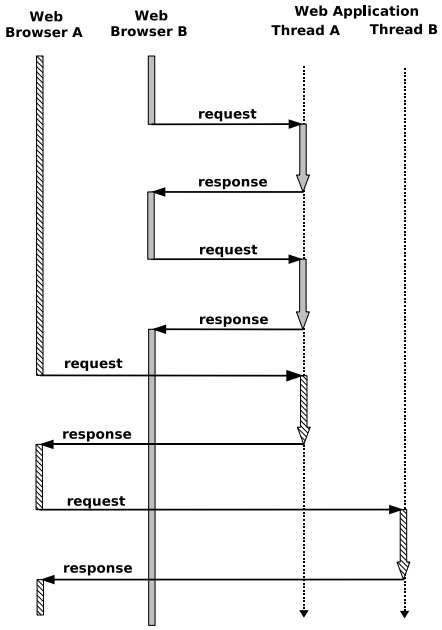

図 2 keep-alive タイミングとスレッドの処理
Fig. 2 Keep-alive timing and thread execution.

2.2 web アプリケーション走行プラットフォームの特徴

テスト環境用プラットフォームは，内部にスレッドを生成して web スクリプトを実行させるタイプの単一プロセス構造の web サーバである．これに対して，

☆ http://openqa.org/selenium-ide/
☆☆ ハエトリソウの学名 Dionaea Muscipula に由来する．
☆☆☆ いわゆる “CGI スクリプト” と同義で使う．

運用（本番用）web サーバには多種多様な形態があるが，同時リクエスト量に対するプロセスの数の増加によるスケーリングが共通する特徴である．多重アクセスに対するプロセスの選択ならびに，その内部におけるスレッドの調整と同期は，前置プロセスならびに，低位ライブラリやデータベースで行われ，web スクリプトの実行は，基本的にはどの後置プロセスのスレッドでも実行できるようになっている☆．

2.3 新規デバッグ機能の具体案

単一セッションの通しデバッグと相互作用のある複数セッションの同時デバッグ☆☆により，複数スレッドが交錯する環境でのデバッグを想定する．

- たとえば，2 つのブラウザを同時に使ってそれぞれのセッションをインタリーブさせたデバッグをしたいとする．実施者は 2 人いるか，ブラウザシミュレータを動かしておく．この場合，スレッドのブレークがどのセッションに対応しているかすぐには分からない．ブレークしたスレッドの ID と session ID の対応関係を知る必要があるが，スレッドアウェアだけでは難しい．
- 同様に，2 つのブラウザを同時に使ってデバッグし，セッションの 1 つはブレークさせてステップ実行させ，他方はブレークさせずに，スキップさせたいとする．この場合，両方ブレークされては混乱する．そのため，セッションごとに有効なブレークポイントという新機能，つまり 1 つのスレッドが実行するセッションは変化するから，このブレークポイントにヒットしたときは，指定のセッションに限ってブレークが発生する，という機能が必要である．
- ブラウザを 1 つだけ使用しデバッグする．だが，そのリクエストを処理するスレッドは 2 つ以上あるかもしれない．運用環境用サーバでは，それらのスレッドは異なるプロセスで実行される可能性が大きい．よって，全プロセスにブレークポイント設定をしなければならない．この操作は面倒である．そもそもプロセスはリクエスト発生のタイミングで生成する可能性があるから，開発者には設定が不可能に近い．よって，タイムリな自動的ブレークポイントの設定機能が必要である．
- あるブラウザに着目する．対応するセッションの実行全体をステップするのは効率が悪い．セッション開始時にはブレークするが，その後デバッグしたいスクリプトの直前のページが現れるまでブレークをオフにして進み，そのとき，ブレークポイント設定をオンにして，スクリプトをブレークさせたい．よって，自動ブレークポイント設定のオン・オフ機能が必要である．

上記の要求条件に応えるためには，次章で述べるようなスレッドアウェアデバッグ環境では十分ではない．よって，以下のような新機能を提案する．

(1) **デバッガによる session ID の自動取得とスレッドのグループ化表示：**スレッド群のプロセス ID によるグループ化に加えて，session ID によるグループ化も行い表示する．

(2) **セッションを指定した形での，アクションの起動時におけるブレークポイントの設定およびそのオン・オフ：**一般的なフレームワークは “モデル・ビュー・コントローラ（MVC)” 構造を提供し，その枠組を利用して，web アプリケーションを実装する．よって，デフォルトのブレークポイントはスクリプトの実効上の開始点であるコントローラクラスのインスタンスメソッド，つまり Rails ではアクションが適当である．

上記機能を多様な走行プラットフォームの形態に依存しない（透過な）ユーザインタフェースで提供する．

2.4 提案機能の適用例

図 3 は，書籍購入用の Depot アプリケーション[9]を，上記提案のデバッグ環境を用いてデバッグ操作をしている状況を示す．上段はブラウザの画面を，下段はデバッガのビューの一部を表し，矢印で推移を示す．このアプリケーションは，買い手セッションと管理者セッションからなる．まず，前者をデバッグするものとする．また，そのときの session ID によるグループ化表示を**図 4** に示す．

(1) 上記のセッション指定のブレークポイント設定機能をオンにし，カタログページの “add to cart” のリンクをクリックすると，スクリプトの実行がアクション “StoreController#add_to_cart” でブレークする．

(2) デバッガ上でステッピングなどによりデバッグを進めると，アクション “StoreController#display_cart” へリダイレクションが起こることが確認できる．

(3) さらに，スクリプトの実行を終えるまでデバッグ操作を続けると，カートページがブラウザに表示される．

(4) ここで，ブレークポイント設定機能をオフして，“continue shopping” のリンクをクリックする

☆ シェアナッシングアーキテクチャ[9] と呼ばれる．

☆☆ たとえば管理者セッションと買い手セッションの相互作用，決済時の在庫引き当てのような買い手セッションの相互作用など．

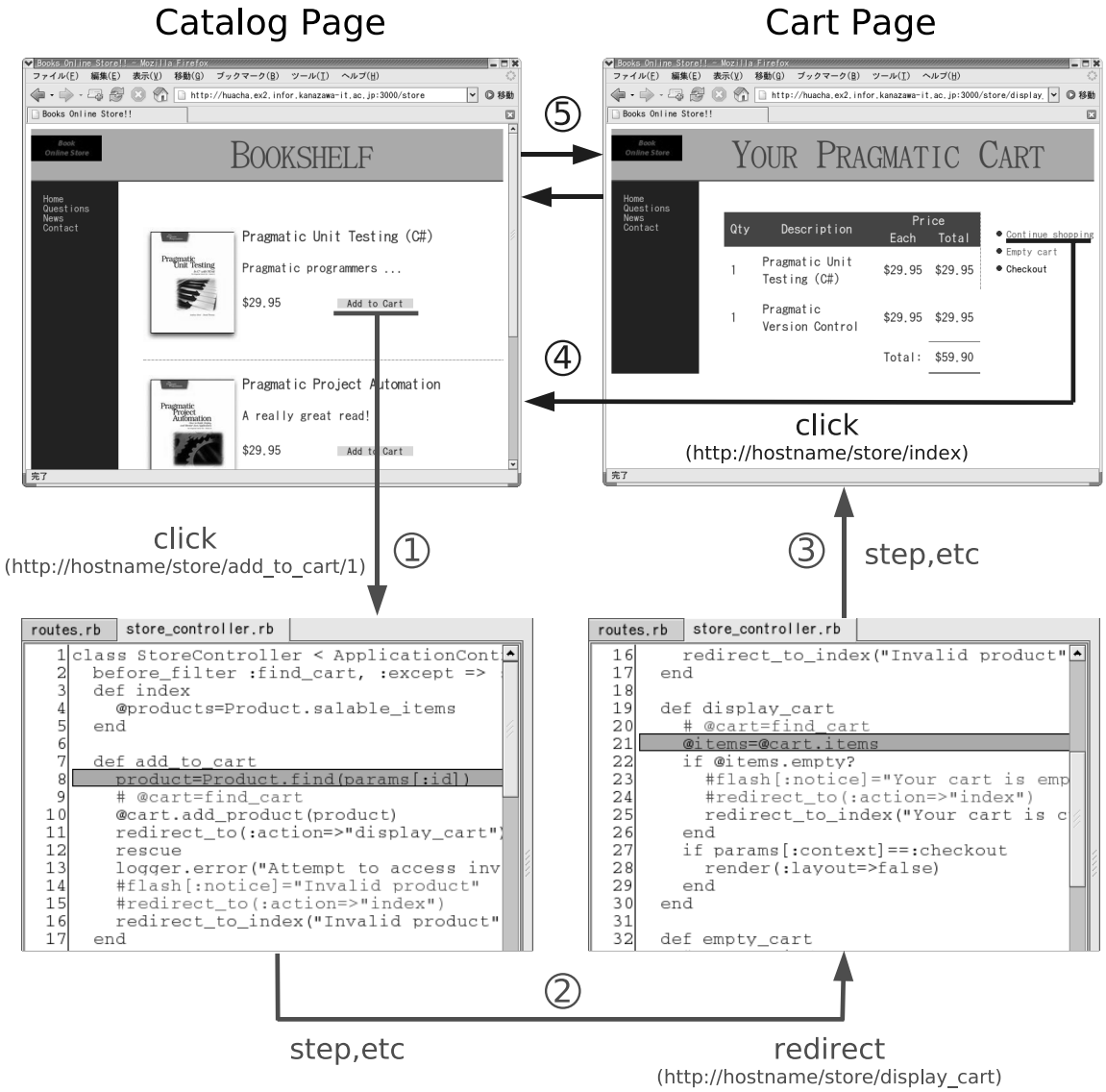

図 3 コントローラアクションでの自動ブレーク
Fig. 3 Automatic breaks at controller actions.

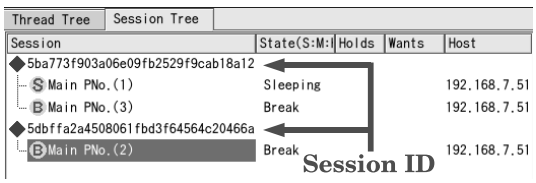

図 4 session ID によるスレッドのグループ化表示
Fig. 4 Displaying threads grouped by session ID.

と，アクション “StoreController#index” は実行されるが，デバッガ上ではブレークはスキップされて，直接，カタログページが表示される．

(5) 上記 (4) の操作を繰り返すとブレークは起こさずにページは互いに切り替わる．

上記 (5) の操作を繰り返している間に，たとえば同じ商品が重複してカートに登録されているというような問題を発見したとする．この場合，再び上記 (1) から (4) の操作を繰り返すと，すべてのアクションでブレークが起こるので，ステップなどにより，問題箇所を特定する．問題箇所を発見し修正したら，カートをリセットして，そのまま，デバッグ作業を続行する．ソースビューには修正したコードが表示される．

“checkout” リンクをクリックした後の注文時に，発注伝票が正しく作成されたかをチェックするためには，もう 1 つのブラウザを用いて，管理者セッションを同

時に試験する．まずは，管理者セッションのブレーク設定をオフにしておく，デバッガは新たなセッションごとにグループ化したスレッドを表示するが，上記の買い手セッションのデバッグ操作には影響がない．問題があれば，ブレーク設定機能のオン・オフをユーザセッションから管理者セッションに切り替えてブレークをオンにしてデバッグを行う．両方のセッションに対するブレークをオンにしておけば，両者のデバッグをインタリーブすることができる．

3. 既存 Dionea のデバッグ機能と実装

2 章で提案した機能を，既開発のデバッガ Dionea [7),8)] をベースに実装する．以下，前者を “既存 Dionea”，バージョンアップ版を “web 対応版 Dionea” と呼んで区別する．

3.1 Dionea の特徴

“既存 Dionea” は，非同期世界のデバッグ環境[5),6)] を意図したデバッガである．つまり，外部とのインタラクションを持つ対象プロセスを停止させずに，デバッグならびにモニタリングするツールを意図していた．これを，既存の high-intrusive 型デバッガ☆に対比して，low-intrusive 型と呼んでいたが，スレッドのスナップショットの非同期表示も含めれば，非同期型（asynchronous）スレッドアウェアと呼ぶのが適当である．また，複数のプロセスが対象であることから 1 対多非同期結合型（one-to-many asynchronously coupled）と表現している．

3.1.1 複数プロセスの同時リモートデバッグ

図 5 に示すように，リモートで走行しているプロセスを捕捉することと起動することができる．以降，デバッグ対象プロセスをデバッギ，デバッガをデバッグクライアント，デバッギ中のデバッグエージェントをデバッグサーバと呼ぶ．

“web 対応版 Dionea” では，図 5 の上側に示すように，プロセスによりグループ化したスレッド群に，session ID によるグループ化構造を重畳する（詳細は 4.4 節を参照）．

デバッグサーバはデバッギのコードをロードし，デバッグイベント生成用フックを設定してから，同一プロセス内でデバッギを走行させる．“既存 Dionea” では，ユーザ操作により，デバッグクライアントから，走行中のデバッグサーバに接続（connect）するか，直接起動（launch）しこれに接続するという手段で，デバッギプロセスをデバッグ状態にする☆☆．これを “プロセスの捕捉” と表現する．“web 対応版 Dionea” では，個別の web サーバに対して，web スクリプトを実行するためのプロセスが生成するタイミングで，この捕捉動作を自動的に起こすことにより，個別の形態に透過なユーザインタフェースを提供する．その具体的対策は，4.2.4 項で述べる．

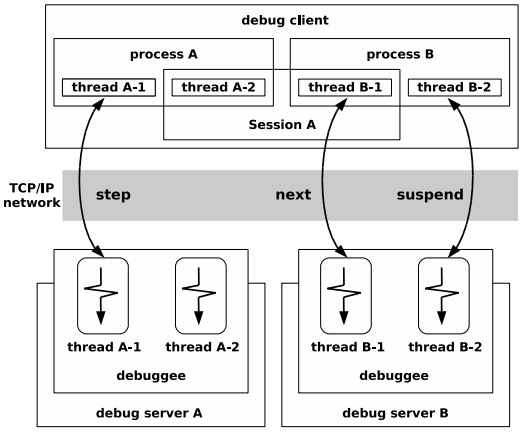

図 5 Dionea 概観
Fig. 5 Dionea architecture.

3.1.2 low-intrusive なスレッド操作

デバッギプロセス中の任意のスレッドを，他のスレッドに影響せず，デバッガで捕捉する機能である．これは，ブレーク時やコマンド投入時にデバッギプロセス全体が凍結される high-intrusion 環境と対比できる．このため，コマンド・レスポンスの送受信には，すべて non-blocking な TCP ソケットを使用している．

ブレークしたスレッド以外は走行を続けるため，同時に複数のスレッドがブレークすることは一般的に起こる．このとき，どのスレッドを選択するかは開発者に委ねられる．また，走行中の任意のスレッドに対して外部から中断（suspend）できることも特徴である．

この機能は，複数セッションの同時デバッグとデバッギの内部状態の非同期モニタリングにも有効であるため，“web 対応版 Dionea” でもそのまま継承する．

3.1.3 disturb モード

未知なあるいは開発者には特定することの難しいスレッド群の捕捉を，デバッグサーバへ要求する機能を用いて，開発者によりオン・オフの設定ができる．当初は，新規生成のスレッドをブレークするための disturb モードを考案した[7)] が，これに，リモー

☆ GNU GDB, Java 用の “jdb”, Ruby 用の “ruby-db”, Python 用 “Pdb” など既存デバッガはどれも high-intrusive である．

☆☆ デバッグ状態解除（disconect）と再接続（re-connect）も可能である．

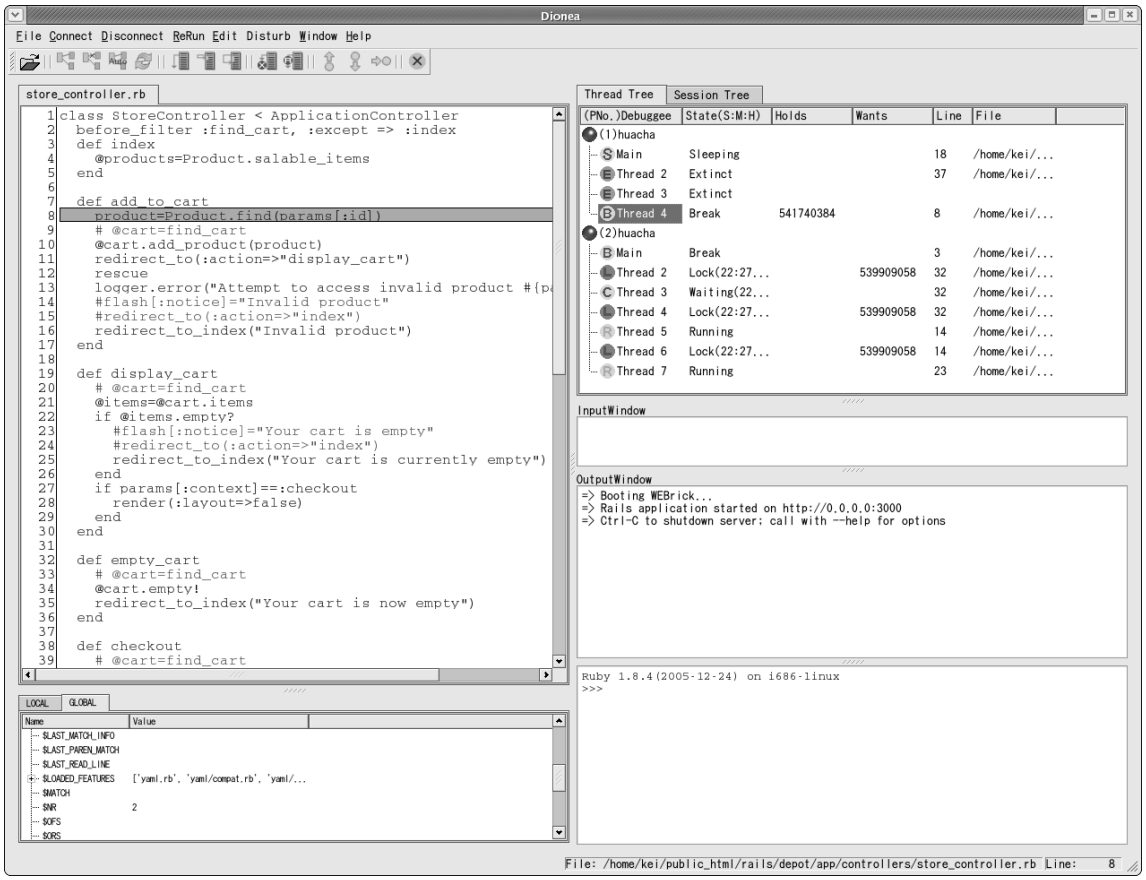

図 6 ユーザインタフェース
Fig. 6 User interface.

ト呼び出し（RMI）で生ずるプロキシをブレークするための disturb-rmi モードを追加した[8]．“web 対応版 Dionea” では，さらに，“指定セッションのアクション起動に対するブレークポイント設定をオンにするため”，disturb-action モードを追加する（詳細は 4.5.2 項を参照）．

3.2 ユーザインタフェース

ユーザインタフェースは 6 つのビューとツールバーからなる GUI である．これを**図 6** に示す．

デバッギビュー（図の右上）：捕捉したデバッギプロセスをノードとしスレッドを葉とするツリー形式で非同期スナップショットを表示[8]する．スレッドの状態を Run，Extinct，Break，Locked，Conditional wait，Sleeping で表示し，これに，状態変化時刻，保持・要求ロックオブジェクト，停止位置などを付加している．現在選択しているスレッドを青色で強調してある．“web 対応版 Dionea” では，このツリーの裏側にタブを配置して，session ID でグループ化したスレッドのツリーを表示する（詳細は，4.4 節を参照）．

スレッドコンテクスト：以下のビューをスレッドコンテクストと呼び，デバッギビューの葉のクリック選択により表示が切り替わる．

- **ソースコードビュー（図の左中）：**選択されたスレッドが実行しているソースコードを表示する．ソースコードが修正されているときは，再取得してつねに最新のコードを表示する．ブレーク位置は黄色でハイライトする．
- **コマンドシェル（図の右下）：**デバッグコマンドの入出力を行うための CUI である．
- **変数ビュー（図の左下）：**グローバル変数とローカル変数の値をオブジェクトツリーの形で表示する．

プロセスコンテクスト：プロセスにまたがるスレッドコンテクストの切替えにともない表示が切り替わる．

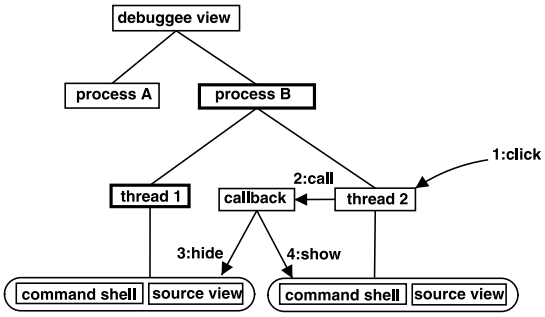

図 7 ウィジェット構成
Fig. 7 Widget architecture.

- **入出力ビュー（図の右側中 2 つ）：**デバッギの標準入出力であるが，web アプリケーションには使用されない．
- **ブレークポイント編集用ダイアログウィンドウ：** ツールバー（図の上段）のアイコンからウィンドウを開いて，ブレークポイント（メソッドとデータへのブレークポイント，ポイントカット）とトレースポイント（トレースコードを付加するとき）を設定する．行への設定はソースコードをクリックする．

ツールバー： 各種ダイアログ用のアイコン（ソースコード取得用，ブレークポイント設定用，connect/disconect 用），ワンタッチコマンド（continue/step/next，suspend，rerun，up/down，quit）がある．“web 対応版 Dionea” では，ここに配置した disturb モードのオン・オフ用のポップアップメニューに “disturb-action” モードを追加する（4.5.2 項の図 21 参照）．

3.3 Dionea の実装

3.3.1 デバッグクライアント内部

デバッギプロセスとスレッドは，**図 7** に示すように，PyQt のウィジェットの階層構造で管理し，表示を切り替える．“web 対応版 Dionea” では，session ID によるスレッドのグループ化構造を追加して，表示する（4.4 節の図 16 と図 17 を参照）．

3.3.2 デバッグサーバ内部

図 8 に，Python 用デバッグサーバの全体構成を示す．デバッグサーバのコードは，インタプリタが提供するコールバック API であるトレースフック設定用のラッパモジュール（“ttc”），コマンドリスナおよびコールバック関数群よりなる．Ruby 用のデバッグサーバでは “ttc” 部分が不要である．コマンドリスナは，デバッグコマンドの受信とデバッグイベントの送信を行うための専用スレッドである．

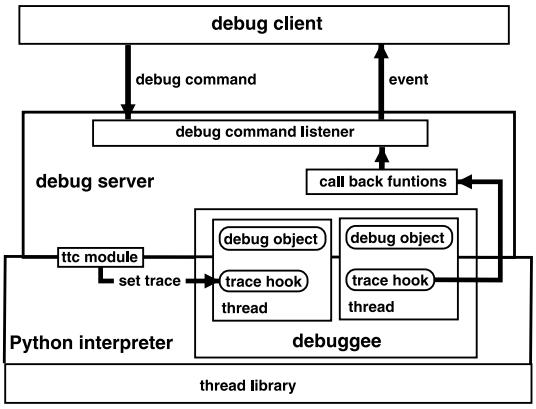

図 8 Python 用デバッグサーバ概観
Fig. 8 Debug server architecture.

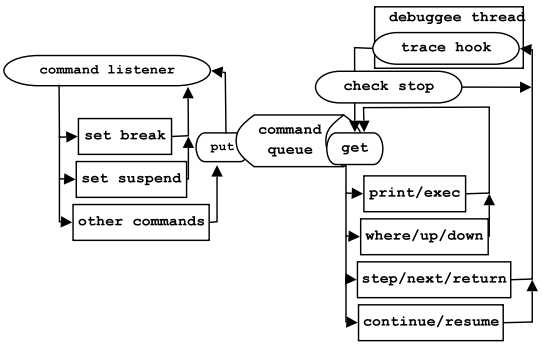

図 9 コマンドの流れ
Fig. 9 Command dispatching.

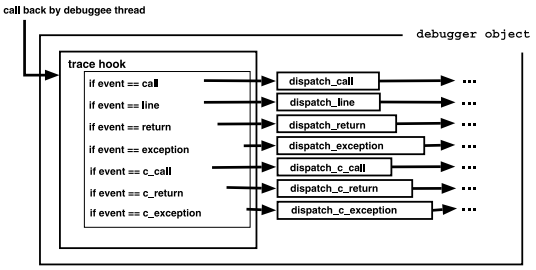

図 10 トレースフックの呼び出し
Fig. 10 Call back by debuggee threads.

フックの設定： フックは，**図 10** で示すトレースフックでは生成できないデバッグイベントを生成するためにデバッギに埋め込むトレースコードである．デバッグサーバはデバッギをロードしフックを設定してから走行させる．“web 対応版 Dionea” では，この設定に，session ID の自動取得ならびにアクションへのブレークポイント設定のためのフックを追加する（実装の詳細は，4.3 節を参照）．

ブレークポイントの設定：図 9 の左側に示すように，

コマンドリスナが設定する．その他のコマンドはスレッドごとに実体化した生産者・消費者キューに送られ，デバッギスレッドにより実行される．

デバッグ API によるイベントとコールバック関数によるブレークの検出： 図 10 に示すような，インタプリタ内部で起こるイベントに対して，インタプリタはデバッギスレッドからの暗黙の呼び出し（コールバック）を発生させる．ブレークのヒットは，このコールバックの中で検出する．図 9 の右側にコールバック発生時の処理手順を示す．現在は，オーバヘッド抑制策として，ライブラリ実行中のコールバック時の判定はスキップしている．

4. web アプリケーション向けの機能の実装

4.1 既存 Dionea をベースとした実装

2.3 節で提案した機能を実現するため，既存 Dionea に施した実装を，以下に示す．

(1) **session ID の自動取得：** デバッグサーバがデバッギのライブラリに設定するフックを追加する．このフックをヒットしたスレッドが session ID を取得してデバッグクライアントに対して通知する．

(2) **session ID によるスレッドのグループ化表示：** デバッグクライアントの持つスレッドのスナップショット管理・表示機能に session ID ごとにグループ化したものを追加する．

(3) **セッションを指定した形でのブレークポイントの設定：** 指定セッションのスクリプト起動に対して，アクションに 1 度だけ有効なブレークポイントを設定するためのフックを，Rails ライブラリに設定する．

(4) **上記 (3) の設定のオン・オフ：** この設定を有効化・無効化するために新しい disturb モードである，disturb-action モードを追加する．また，デバッグクライアントから，接続しているすべてのデバッグサーバに対して，disturb-action モードをオン・オフ指示するためのコマンドを追加する．ユーザインタフェースのツールバーにこのコマンドを送出するメニューを追加する．

上記の機能を，web サーバの形態に対して透過なユーザインタフェースとするため，デバッグクライアントとデバッグサーバの手動接続機能を自動化して実装する．典型的な形態[9),10)] に対する対策は次節で述べる．

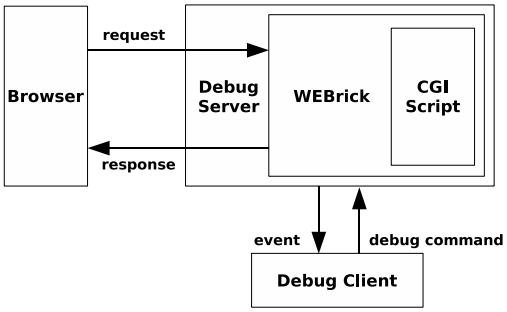

図 11 WEBrick 使用時の Dionea 起動方法
Fig. 11 How Dionea runs with WEBrick.

4.2 web 対応版 Dionea の起動とデバッグサーバへの自動接続

4.2.1 テスト用 web サーバ（WEBrick）を使う形態

WEBrick は，Ruby 標準ディストリビューションに含まれる web サーバであり，テスト用か小さなサイトに限られ使用される．この種の web サーバは複雑な設定もなく起動は容易である．デバッグするときは，デバッグサーバに WEBrick を起動させればよい．**図 11** は，テスト用サーバ（WEBrick）を使用するときの Dionea の走行を示す．

手順は一般の Ruby スクリプトをデバッグする場合と同様，次のように，Rails の “script/server” スクリプトを引数としてデバッグサーバを起動する．

```
$ dioneas.rb [ -p ⟨port number⟩]
    /path/to/rails/script/server
```

これにより，デバッグサーバ内部の “main” メソッドが実行される．

4.2.2 運用サーバ使用：Apache に mod_ruby モジュールを組み込む形態

デバッグサーバを前置フィルタとして駆動する．

図 12 は，Apache と mod_ruby モジュールを使用するときの Dionea の走行を示す．この場合，mod_ruby モジュールがデバッグサーバを起動し，デバッグサーバがデバッギをデバッグ状態で走行させる．

デバッグサーバには，(main とは別の) “mod_ruby からのエントリ☆” を追加してある．ここで，disturb モードの設定も行う．Rails と Apache の組み合わせるときのコンフィギュレーションは，ローカルな “.htaccess” ではなく “httpd.conf” に設定するが．その内容

☆ Apache::RubyRun#handler,
Apache::RailsDispatcher#handler

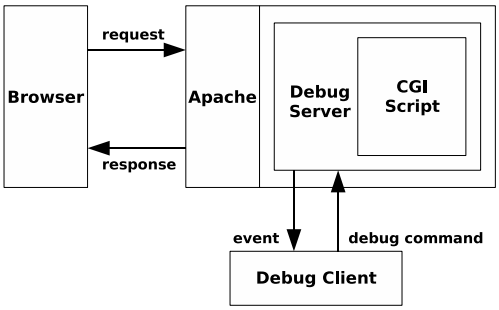

図 12 Apache（mod_ruby，mod_python）使用時の Dionea 起動方法
Fig. 12 How Dionea runs with Apache with mod_ruby or mod_python.

は文献 8) とほぼ同様である☆1.

4.2.3 運用サーバ（Apache と FastCGI プロセス）を使う形態

静的なリクエスト（単純なファイルの取得）はすべて Apache が引き受け，FastCGI は動的リクエスト（web スクリプトの実行）のみ扱う．起動時間を短縮するため，FastCGI は，Ruby インタプリタ，フレームワークのライブラリ，データベースへのコネクション，および web アプリケーションのコードをロードしたまま走行を持続する．大規模なライブラリを用いる Rails は FastCGI を推奨している．

図 13 は，FastCGI を使用するときの Dionea の走行を示す．最初の HTTP リクエストが到着すると，Apache は FastCGI プロセスを起動する．FastCGI に（CGI として）デバッグサーバを起動させ，デバッグサーバが web スクリプトをデバッグ状態にする．

デバッグサーバ内部には FastCGI からのエントリを追加してある．コンフィグレーションファイル（“httpd.conf” または “.haccess”）には，FastCGI 用

☆1
```
⟨IfModule mod_ruby.c⟩
RubySafeLevel 0
RubyRequire rubygems
RubyAddPath /path/to/dionea
RubyRequire /path/to/dioneas
⟨Location /depot⟩
SetHandler ruby-object
RubyHandler Apache::RailsDispatcher.instance
RubyTransHandler Apache::RailsDispatcher.instance
RubyOption rails_uri_root /depot
RubyOption rails_root /path/to/rails
RubyOption rails_env development
⟨/Location⟩
⟨/IfModule⟩
```

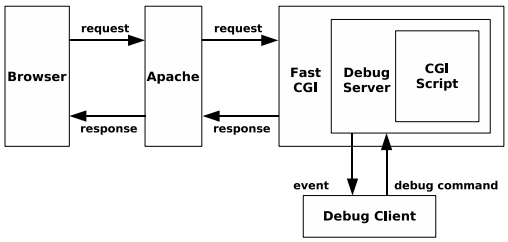

図 13 Apache（FastCGI）使用時の Dionea 起動方法
Fig. 13 How Dionea runs with Apache and FastCGI.

の設定[10] をする☆2．Rails を呼び出すスクリプト “dispatch.fcgi” には，Dionea のデバッグサーバの追加部分を呼び出す行☆3を追加する☆4.

4.2.4 デバッグクライアントによるデバッグサーバの自動捕捉

デバッグクライアントは，デバッグサーバの所在，つまり，ホスト名（あるいは IP アドレス）とコネクション用ポート番号をあらかじめ知ることができない．このため，後者が前者にホスト名（IP アドレス）とポート番号を通知する．この通知を行うためには，後者は前者の所在，つまり，ホスト名（IP アドレス）と通知用ポート番号を知る必要があるが，これは Dionea のコンフィギュレーションファイルで指定することにしている．開発者がダイアログを通じて通知用ポート番号を指定すると，デバッグクライアントは “自動接続モード” になる☆5．これにより，デバッグクライア

☆2
```
⟨VirtualHost *:8080⟩
DocumentRoot /path/to/rails/public
ErrorLog /path/to/rails/log/server.log
AddDefaultCharset UTF-8
⟨Directory /path/to/rails/public⟩
AddHandler fastcgi-script .fcgi
AddHandler cgi-script .cgi
Options +FollowsymLinks +ExecCGI
RewriteEngine On
RewriteRule ^$ index.html [QSA]
RewriteRule ^([^.]+)$ $1.html [QSA]
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^(.*)$ dispatch.fcgi [QSA,L]
⟨/Directory⟩
ErrorDocument 500 ”⟨h2⟩Application error⟨/h2⟩Rails
application failed to start properly”
⟨/VirtualHost⟩
```

☆3 require ”/path/to/dioneas”
☆4 これは暫定解である．dioneas.rb から dispatch.fcgi を起動する方がよいが，何らかの要因で “dispatch.fcgi” というファイル名が固定しているようである．
☆5 PyQt の QServerSocket クラスを使い connect 要求受付けサーバになる．

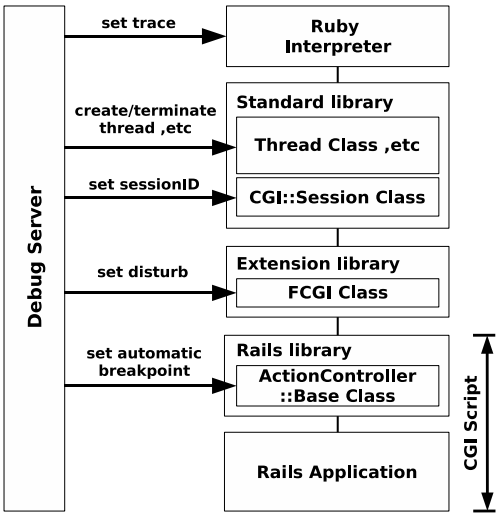

図 14 デバッグサーバによるフックの設定
Fig. 14 Hooks set by debug server.

ントからデバッグサーバへの接続が自動的に起こる．

4.3 フックの設定

図 14 は，Ruby ライブラリへデバッグサーバが設定するフック全般を示す．上から 2 つはスレッドアウェアのための既存のフックで，残りは web スクリプト用に追加したフックである．3 番目は session ID 取得用，4 番目は web スクリプト起動時に disturb モードをオンするためのフック，5 番目はセッションごとのブレークポイントをアクション起動時に設定するためのフックである．

4.4 session ID の自動取得

4.4.1 session ID を通知するフックとデバッギビュー

デバッグサーバは，**図 15** に示すようなフックを Ruby 標準ライブラリ☆に設定する．これをヒットしたスレッドが session ID を知り，デバッグクライアントに通知する．

個々のスレッドはセッションを変える．これを表示するため，ユーザインタフェースのデバッギビューには，スレッドビューとともにセッションビューが追加してある．これを，**図 16** および**図 17** に示す．また，2 つ以上のセッションが存在すると，ブレーク状態（B で表示）のスレッドは複数になることも考慮して，ラン状態（R で表示）からブレーク状態になったスレッドは点滅表示して開発者の反応を促す．開発者がこれをクリックすると図のような青色表示に変わり，実行中のソースコードがソースビューに表示される．現在選

☆ CGI::Session クラスのコンストラクタ initialize メソッド．

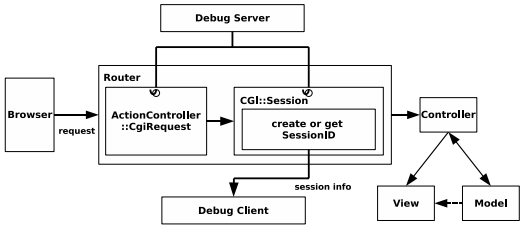

図 15 セッション ID を取得するためのフック
Fig. 15 Hooks to get session ID.

択されているセッションに対しては開発者の反応を待たずに自動的に表示を変える（効果と問題点は 7.2 節で述べる）．

4.4.2 テスト環境用 web サーバにおけるセッション

WEBrick を使用した場合には，セッションビューは，図 16 のように表示される．この場合は，3 つのブラウザを使用しているため，セッションは 3 つ生成されている．静的リクエストを処理するスレッドなどは，スレッドビューには表示するが，セッションビューには表示しない．

デバッグクライアントの内部では，セッションは，**図 18** のような二重の階層構造で管理されている．

4.4.3 運用環境用 web サーバにおけるセッション

FastCGI を使用したときは一般にデバッギプロセスは複数できる．どのプロセスが HTTP リクエストを処理したとしても，図 15 で示すフックがセッション ID とスレッド番号を通知してくれる．このため，テスト用サーバと同一の形式でセッションビューの表示ができる．図 17 の例は，2 つのブラウザを同時使用したときのセッションビューである．スレッド表示にはプロセス番号も付記してある．最初のスレッド（S 状態）は，web スクリプトの実行を終えて，次のリクエストを待っている．セッションは変わるかもしれない．もう一方のスレッド（B 状態）は現在のリクエストを実行中にブレークしている．現在デバッガが選択しているのは 3 番目のスレッドで，もう 1 つのセッションを実行中にブレークしている．

このとき，デバッグクライアント内部では，セッションは**図 19** に示すような認識がされている．

4.5 セッションを指定した形でのブレークポイント設定

4.5.1 アクションへのオン・ザ・フライなブレークポイント設定

動的リクエストが来ると web スクリプトは最初の行から実行を開始する．Rails のスクリプトでは，次

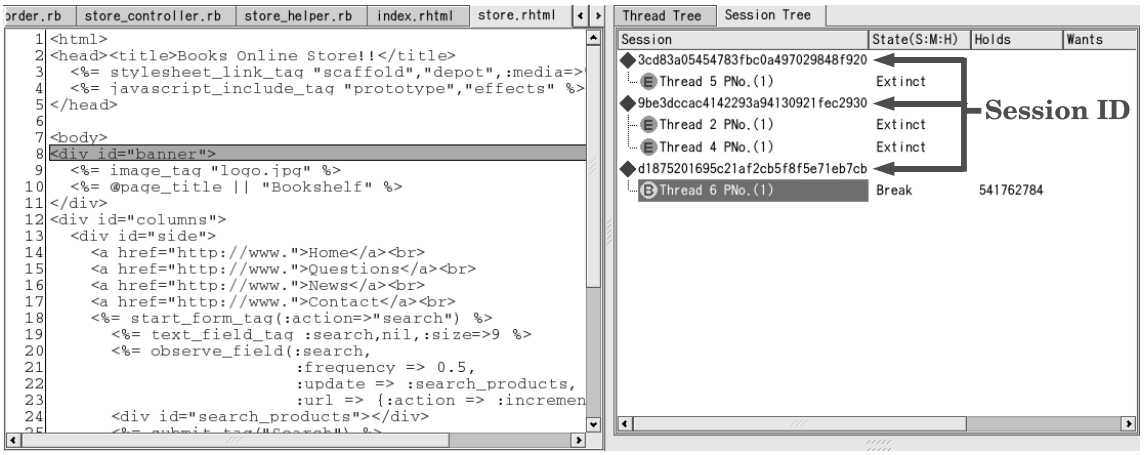

図 16 セッションビュー（WEBrick 使用時）
Fig. 16 Session view for web application threads (using WEBRick).

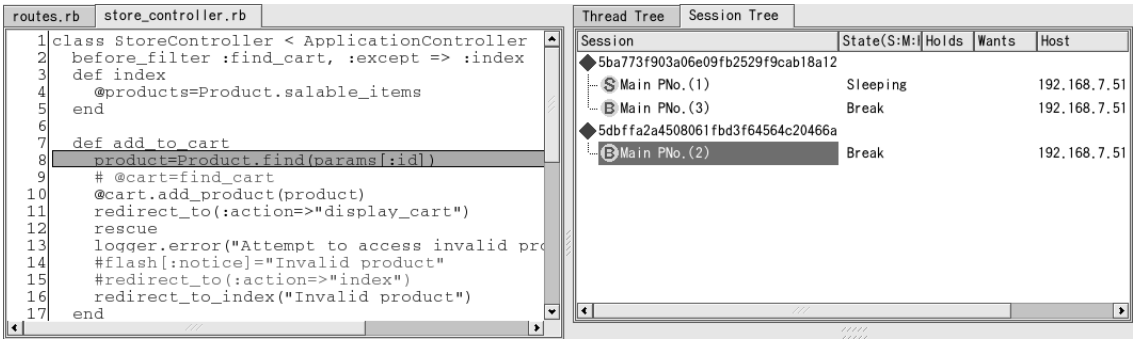

図 17 セッションビュー（FastCGI 使用時）
Fig. 17 Session view (using FastCGI).

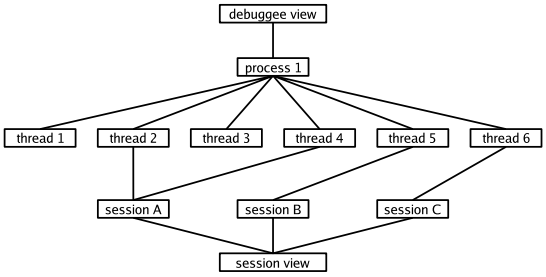

図 18 セッションによるスレッドのグループ化（WEBrick 使用時）
Fig. 18 Grouping threads by sessions (using WEBRick).

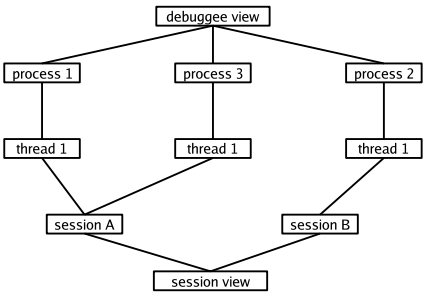

図 19 分散スレッドのセッションによるグループ化（FastCGI 使用時）
Fig. 19 Grouping distributed threads by sessions (using Apache/FastCGI).

に開始スクリプト（“router” と呼ばれる）が，コントローラクラスを実体化し，アクションに対応するインスタンスメソッドを特定して呼び出す．よって，ブレークポイントの設定はこのタイミングで毎回行う．このため，デバッグサーバは，Rails ライブラリの中の，URI をアクションへマッピングするメソッドに，**図 20** に示すようなブレークポイント設定のフックを設定する．このフックにより設定されるブレークポイントは 1 度だけ有効なテンポラリなものである．

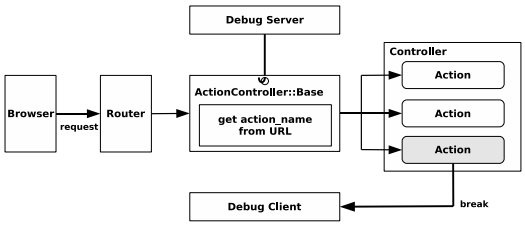

図 20 自動ブレークポイントを設定するためのフック
Fig. 20 Hooks to set breakpoints automatically.

4.5.2 disturb-action モードとその有効化と無効化

disturb モードの一種として disturb-action モードを加える．

- **ワンタッチのオン･オフ：** disturb モードは，**図 21** に示すようなポップアップメニューでオン・オフする．
  この操作により，対象デバッギに次のようなコマンドが送出される．
  `>set-disturb-action-mode true/false` ⟨*session ID*⟩
  この session ID は現在選択している（スレッドの属する）セッションを表す．disturb-action モードのオン・オフは捕捉しているデバッギ全部に向けて送出する．
- **新規生成のデバッギに対する disturb-action モードの伝播：** disturb-action モードは，(Apache や lighttpd により）新規に生成されるデバッギプロセスへ伝播させなければ，セッション指定のブレークポイント設定ができない．このため，connect コマンドを次のように拡張する：
  `>connect` ⟨*session-info list*⟩
  ここで，セッション情報リスト（session-info list）は，現在捕捉しているすべての session ID とそれぞれのセッションに対して disturb-action モードが有効か無効かを示すフラグからなる．このコマンドは，デバッグクライアントが自動的に送出する．
- **disturb モードの初期設定：** デバッグサーバ内部における disturb モードの決定手順を**図 22** に示す．初期状態における disturb モードは，既知のセッションごとに，disturb モードまたは disturb-action モードのいずれかが有効になっている．未知のセッションについては disturb モードが有効になる．Rails のアプリケーションは，最初は disturb モードであるが，1 度 disturb-action モードが設定されると，disturb モードには戻さない．

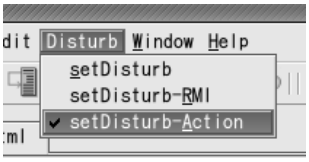

図 21 disturb-action モードの設定と解除
Fig. 21 Set and reset disturb-action mode.

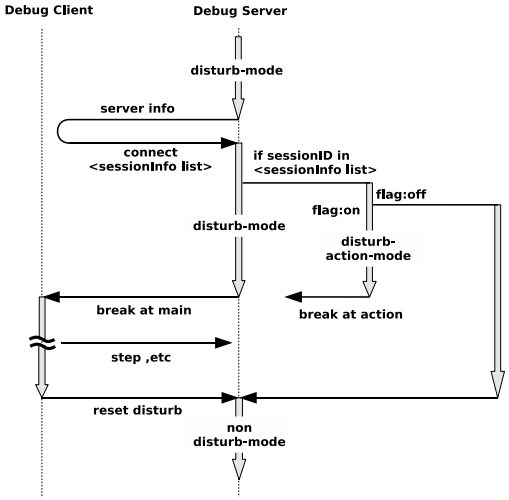

図 22 デバッギ起動時の disturb-mode 設定方法
Fig. 22 How disturb-modes are set at starting a debuggee.

5. 適用シナリオ

5.1 セッションを使用する web アプリケーション

5.1.1 複数セッションの相互作用

たとえば，ホテルの予約と座席予約がセットになっているような web 予約システムでは，一般に最後の決済ページでのみ，予約引き当てにトランザクションを使う．このとき二重予約防止策や予約失敗時の対策☆を確認するには，複数の顧客セッションをインタリーブさせてデバッグする必要がある．このためには，両方のセッションの disturb-action モードをオンにし，セッションをステップ粒度で切り替えながらタイミングを作り出すことになる．セッションビューには，スレッドがグループ化表示されているのでこれを操作する．セッション選択はスレッドの選択と連動している．

5.1.2 Ajax コールのセッション内相互作用

Ajax コールはサーバ側からみると，通常の HTTP リクエストと同様に扱われる．このため，複数の Ajax

☆ 決済の段階で “満席の場合があります”，とメッセージを出して，予約ページに戻しロールバックは顧客に委ねるなど．

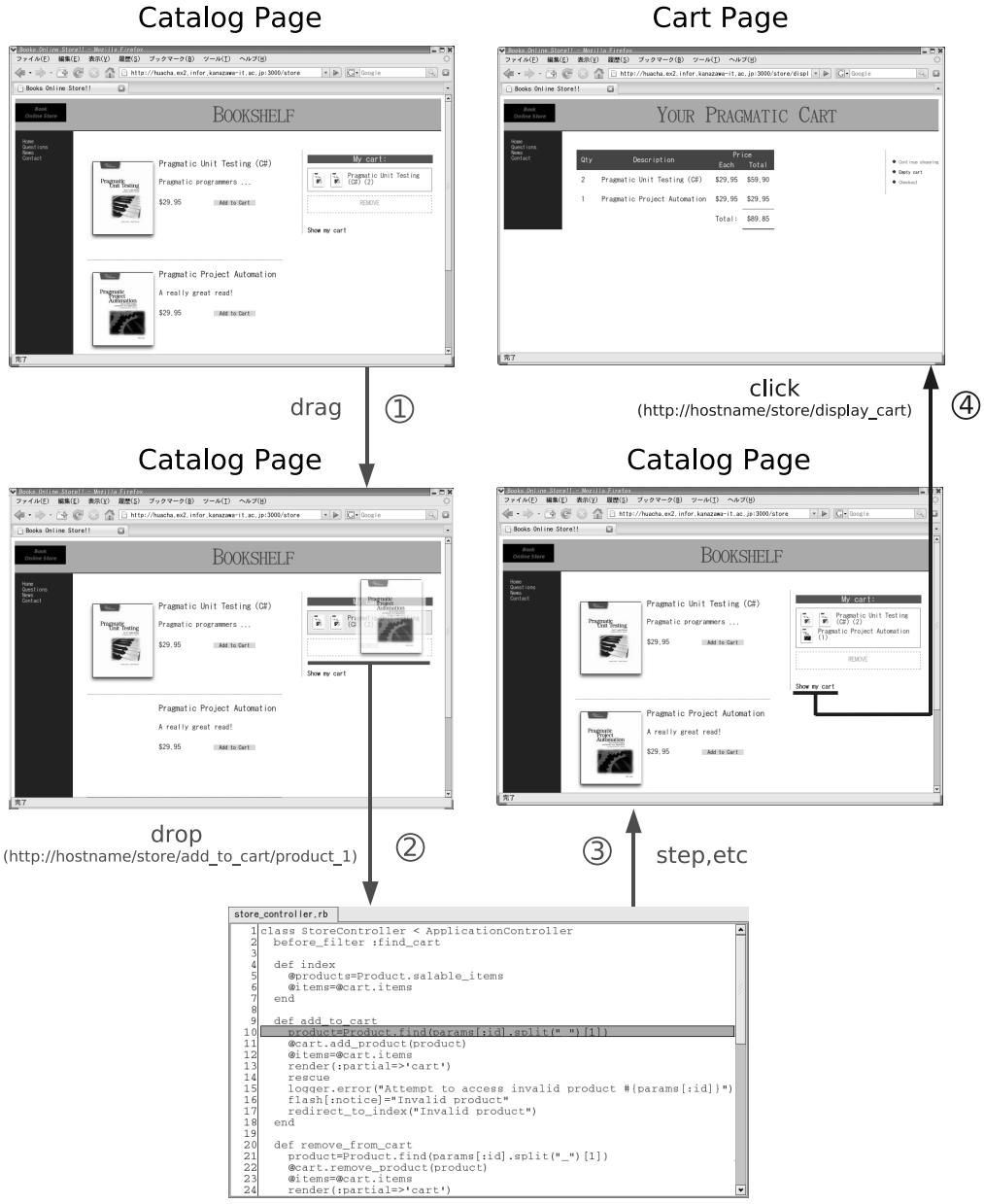

図 23 Ajax コールによるコントローラアクションでの自動ブレーク
Fig. 23 Automatic breaks at controller actions by Ajax.

コールが連続すると，セッションデータなどの競合が起こる可能性がある．この場合，対応するスクリプトをインタリーブさせて，リクエストの処理順序を意図的に入れ替え，競合の原因を究明することができる．**図 23** は，Depot アプリケーションのカート部分を Ajax を用いてビジュアル化した例であり，カタログページにはビジュアル化したカート部分がある．

(1) disturb-action モードが有効な状態で，商品の thumb-nail をカート部分にドラッグしてカート上に移動する．

(2) ここでドロップすると，デバッガ上でアクション “StoreController#add_to_cart” にブレークが起こる．

(3) アクションをデバッグしこれが完了すると，カタログページのカート部分だけが再描画されて商品が追加されることを目視できる．

(4) disturb-action モードを解除して，“show my cart” のリンクをクリックすると，スクリプトはブレークを起こさず実行されて，カートページが表示されカートの内容を確認できる．

たとえば，上記 (2) が終わった時点で，ブラウザ上で別の書籍をカートへドラッグ・ドロップすると，同時に 2 つの web スクリプトの実行がブレークする．以降のデバッグ操作により，どちらかのリクエスト処理を先行させても，インタリーブさせても，いずれか一方が無効となることが確認できる☆．

5.1.3 ログイン画面へのリダイレクト

認証が必要になるページに認証する前にアクセスを試す．認証画面にリダイレクトすれば OK である．アクセスができてしまうというような問題があれば，disturb-action モードを有効にしてスクリプトのデバッグを行う．

5.1.4 ブラウザでの back page への戻りアクセス

たとえば，買い物を決算し注文が完了した後，ブラウザの戻りボタンを操作すると，注文リクエストが再送され，注文が重複してしまうことが起こる．再送リクエストも通常の HTTP リクエストであるため，デバッガは通常の web スクリプトに対すると同様に動作する．たとえば多重送信防止用の確認画面を出しているか，ワンタイムトークン☆☆をうまく管理できているか，などの適切な対策がとられているかをチェックし，問題があればデバッグすることができる．

5.1.5 バグによるスクリプトの実行のクラッシュ

cookie はブラウザに，セッションデータはサーバ側にあるため，ブラウザのクリック操作だけでクラッシュが起きたスクリプトを再起動でき，原因究明のデバッグが可能である．

5.2 セッションを使用しない web アプリケーション

セッションビューには何も表示されない．スレッドビューを使用してスレッドごとのデバッグを行うことになる．初期状態は disturb モードであるから，スレッドはスクリプトのユーザコード開始点でブレークする．

5.3 独自の sesion ID を使用する web アプリケーション

本デバッガは，標準ライブラリの CGI::Session.initialize メソッドにフックを設定して，session ID を取得している．このライブラリを用いている限りでは，session ID は自動取得できる．Rails もこのライブラリを使用している．アプリケーション開発者が，独自の session ID を使用したい場合も，セッション生成メソッド CGI::Session.create_new_id メソッドをオーバライドすれば問題ない．また，Div/Tofu [4] などのように，cookie は使用してもこのライブラリを使用しないで独自管理する場合も，開発者がフック数行を追加すれば，session ID を取得することができ，同様な環境が利用できる．

6. 考察と評価

6.1 web アプリケーションのテストサポート

“Ruby on Rails” は次の 3 段階のテストサポートを提供している．

(1) ユニットテストのためのカスタムアサーション：アサーションの作成と実装を同時進行させる開発方法はテスト駆動開発[12] といわれる．このアサーションをオフラインでの自動テストに用いる．モデル部分（MVC の M）のアサーションをユニットテスト，個々のコントローラのそれを機能テストと呼ぶ．

(2) web サーバのログ解析：web アプリケーション全般に使用されている post-mortem な手段である．

(3) “breakpoint コール” の埋め込み：デバッガの代用である．固定埋め込みのブレークポイントであり，ヒットすると対話型 Ruby（irb）のセッションが開く．

本デバッガは，上記 (3) よりも格段にパワフルな機能を提供する．上記 (1) はリピータビリティがあり非常に便利であるが，コントローラの結合機能テストと GUI のデバッグならば本デバッガが有効である．デバッガで解決した問題は積極的に上記 (1) に反映するとよいが，ブラウザ操作は，Firefox のプラグイン Selenium-IDE を用いて記録することにより，直接再現することもできる☆☆☆．

Rails は，また，以下の開発段階ごとにデータベースを使い分ける．これらの区別を Rails の環境変数

☆ checkout ページでの確認措置があるので，この設計では問題はないが，重要データがある場合は，データベース化によるロック対策が必要になる（この議論は文献 9) 参照）．

☆☆ トランザクショントークンとも呼ばれ，多重送信によるミス，攻撃を防止する方法の 1 つ．サーバ側で，ランダムなトークンを生成・管理し，それをフォームの隠しフィールドに埋め込みレスポンスを返す．クライアントから送信されてきたトークンと管理しているトークンとが一致する場合，リクエストを受け入れ，同時にそのトークンを破棄する．一致しない場合は，リクエストを受け入れない．

☆☆☆ ただし，5.1.2 項に例示したドラッグ・ドロップのイベントは，バージョン 0.8.6 では標準にはサポートしていないが，拡張機能を使って JavaScript を追加することで再現できる．

(RAILS_ENV) に設定する.

(1) test：ユニットテスト用の一時的なデータベースを用いる.

(2) development：ブラウザと web サーバを用いて行う段階で，Rails のライブラリは，修正版ソースコードがあればそれを自動的に再ロードする.

(3) production：完全な運用状態で，コードの再ロードもデバッグもできない.

本デバッガを推奨できるのは，上記 (2) の段階である．上記 (3) の段階に適用するには，1）パフォーマンスギャップ，2）セッション数の増大というスケールギャップ，3）修正コードの自動アップグレードという機能ギャップ，という難点がある.

6.2 複数セッションデバッグの並列実行

3 つのブラウザを同時に用いて複数セッションデバッグの実験を行う.

図 24 は，WEBrick を用いたときのスレッドの生成を示す．上段は，スクリプトはスレッドセーフでないと設定した場合☆1，下段は，スレッドセーフであると設定した場合☆2を，それぞれ示す．メイン（Main と表示）はリクエストを accept し生産する producer，残りの 3 つは web スクリプトを実行する worker である.

上段の場合は，最初のリクエストを実行するスレッドは走行してブレークしているが，2 番目と 3 番目のスレッドは，最初のリクエストの実行が終了するまでは，Rails ライブラリの内部☆3においてロック（L で表示）されていることが観察できる．つまり，Dionea の持つ low-intrusive なスレッド操作機能は，ここでは有効には生かされない．ただし，WEBrick プロセス全体は停止しているわけではなく，さらにリクエストを受け付けられる状態にある.

下段の場合は，low-intrusive な機能が有効に生かされるので，複数セッションのデバッグは並行に実施できる.

図 25 は，FastCGI を用いたときのスレッドの生成を示す．それぞれのリクエストごとに，FastCGI プロセスが生成され並行に実行されることが観察できる．ただし，FastCGI プロセスが一定数を超えると超えた分だけは FastCGI のプロセスマネージャによりリクエストが保留される．つまり，スレッドによるスケーリングはやらない．この場合，“1 対多構成” が有効に

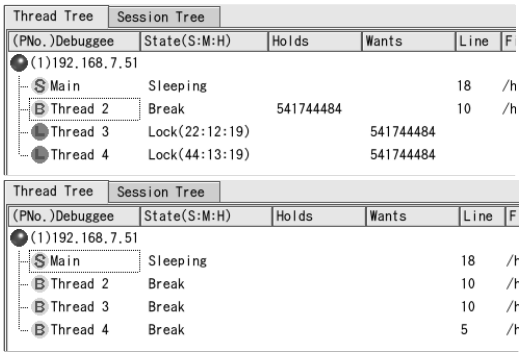

図 24 web スクリプトの並列実行（WEBRick 使用時）
Fig. 24 Concurrent execution of web scripts (using WEBrick).

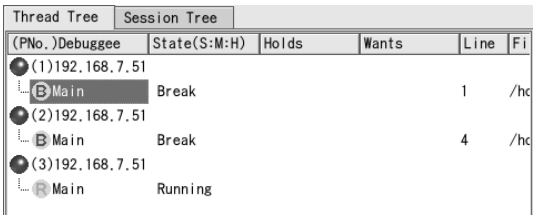

図 25 web スクリプトの並列実行（Ruby 用 FastCGI 使用時）
Fig. 25 Conncurrent execution of web scripts (using FastCGI for Ruby).

生かされ，複数セッションのデバッグは並列に実施することができる.

6.3 性能の問題

デバッグ API による性能低下： Dionea の問題点はデバッギの性能低下である．これは行ごとのコールバック（によりブレークポイントを判定する）という Ruby および Python のデバッグ API に起因している．測定によれば[8]，仮想命令そのものの実行速度は 2 桁近くまで低下してしまう．RMI のスレッドは i/o バウンドであることから 1 桁くらいの性能低下になる[8]．web アプリケーションでは，ブラウザ上のインタラクションが介在するため，この影響はさらに少なくなる．将来的には，コールバックのフック（settrace）を適宜着脱できるようバージョンアップしたいところである．このためには，Ruby のデバッグ API の改良が必要である☆4,[7]．Python に関しては現在の API でも可能性がある.

デバッガの反応速度と快適さ： ソースコードはデバッグサーバより転送されソースビューに描画される．このため最初のブレークに対してソース

☆1 “ActionController::Base.allow_concurrency=false”.

☆2 “ActionController::Base.allow_concurrency=true”．ちなみに，Rails ライブラリはスレッドセーフである.

☆3 DispatchServlet クラス内.

☆4 スタックオブジェクトのリンクの追加など，微妙な点であるが.

コードが描画される速度は，HTML が同一コンピュータ上のブラウザに描画されるよりは少し遅くなってしまう．ボトルネックは転送速度よりも，Qt の canvas へ描画速度である．

7. 今後の課題

7.1 Rails を使用しない web アプリケーションへの対応

アプリケーションが session ID を獲得した時点でセッションビューに表示され，各スクリプトは開始点でブレークする．disturb-session モードを追加実装し，disturb モードまたは該当セッションに対して disturb-session モードがオンであれば，スクリプトは開始点でブレークし，そうでなければスキップ実行させる．これにより，ブレークポイントの設定位置を除けば disturb-action モードと同じ機能が提供できる．

7.2 指定セッションに対するブレークにともなうスレッドコンテクストの自動表示切替え

現在のセッションに関するスレッドがブレークしたとき，ソースビュー（スレッドコンテクスト）表示を自動切替えする機能を実装している．もちろん，GUI で選択されていない session ID を持つスレッドについては，点滅表示にとどめる．ただし，前者は Dionea の実装上例外的☆であるので，Ajax コール多発対策☆☆も含めて一考を要する．

7.3 MVC のビュー部

MVC 構造のビュー（V）部分への自動ブレーク設定は，画面の調整に有効である．実装は，Rails の render メソッドへのフックを追加すれば容易である．

Rails のビューの記述言語である eRuby は，Ruby 同様のステップ実行ができる．しかし，一般に ERB::new(“erb_script”).run などと記述した場合[4]，トレースフックのコールバックからは，ファイル名が取得できないため，もとのソースコードとの対応がとれない．ERB に適当なフックが必要になる．

7.4 Python フレームワーク対応の実装

Django，TurboGears などの最近の Python 向けの web アプリケーションフレームワークの開発が進められておりそのコンセプトは Rails 同様であり，本提案のデバッグ機能を実装するための要件は備えている．つまり，セッション管理用ライブラリを提供し，MVC の枠組みを固定して，URI からアクションへのマッピング規則☆☆☆を持つ．問題は，web サーバと Dionea の接続であるが，ソースコードによる内部構造の解析と走行実験を要する．

8. ま と め

本論文では，web アプリケーション特有の問題に特化したデバッグ環境として，セッションアウェアな機能を提案し，既開発の非同期型スレッドアウェアなシンボリックデバッガ（Dionea）をベースとした実装，および実験による有効性を示した．

本提案の Rails 向け実装（仮称 “Dionea on Rails”）は現時点で一通り完了したこと，また，Rails はその完成度から Ruby 用フレームワークの決定版といえることから，現在は Python 用のフレームワーク TurboGears 向けの実装を行っている．本デバッガの利便性を高めるためには，オープンソースによる実用化が必要であると考える．

謝辞 本論文を改良するために査読者から懇切なご意見をいただいた．また，(個人名は省略させていただくが）研究会での議論，ならびに，阿部倫之氏，山上俊彦氏との私的討論も影響している．

参 考 文 献

1) Acher, D. and Lutz, M.: *Learning Python*, 2nd Edition, O’Reilly (2003). 夏目 大 (訳)：初めての Python 第 2 版，オライリー・ジャパン (2004).
2) Matsumoto, Y., Yamada, A. and Andrew H.: *Programming Ruby: The Pragmatic Programmer’s Guide*, Addison Wesley (2000). 田和 勝（訳），まつもとゆきひろ（監訳）：プログラミング Ruby，ピアソン・エデュケーション (2001).
3) 前田修吾，まつもとゆきひろ，やまだあきら，永井英利：Ruby アプリケーションプログラミング，オーム社 (2002).
4) 関 将利：dRuby による分散・Web プログラミング，オーム社 (2005)
5) Donat, M. and Charl, S.: Debugging in Asynchronous World, *ACM Queue*, Vol.1, No.6, pp.23–30 (2003).
6) Gatlin, K.: Trials and Tributions of Debugging Concurrency, *ACM Queue*, Vol.2, No.7, pp.66–73 (2004).
7) Sato, N., Nagai, K., Itoh, Y., Ogura, M.

☆ 非同期型 Dionea では，必然的にブレークイベントの同時発生が起こるため，スレッドコンテクストの自動切替えは，行わない．例外は，カレントスレッドに対するステップによるブレークとリモートステップ[8] のみである．

☆☆ たとえば Ajax 表示モードの追加．実装は容易である．

☆☆☆ Ruby のアクションが “コントローラオブジェクトのインスタンスメソッド” であるのに対して，Django では “モジュールの関数” である．TurboGears は，CherryPy 内蔵の web サーバを用いるので，アプリケーションが作るオブジェクト階層構造の “callable オブジェクト” にマッピングされる．

and Kosuga, K.: Low-intrusion Debugger for Python and Ruby Distributed Multi-threaded Programs, *IPSJ Journal*, Vol.45, No.12, pp.2741–2751 (2004).
8) 小菅圭介，小倉正充，佐藤規男：Python および Ruby 用 low-intrusion 型デバッガによるピアスレッドの自動認識とその追跡，情報処理学会論文誌：プログラミング，Vol.47, No.SIG 11 (PRO 30) (2006).
9) Tomas, D., Hansson, D.H., Breedt, L., Clark, M. and Schwarz, A.: *Agile Web Development with Rails, A Pragmatic Guide*, Pragmatic Bookshelf (2005). 前田修吾（監訳）：Rails によるアジャイル Web アプリケーション開発，オーム社 (2006).
10) 吉田和弘，馬場道明：ライド・オン・Rails, SoftBank Creative (2006).
11) Ramm, M. and Dangoor, K.: *Rapid Web Applications with TurboGears*, Prentice Hall (2006).
12) Beckm, K.: *TDD: Test-Driven Development By Example*, Addison-Wesley Signature Series (2002). 長瀬嘉秀（監訳）：テスト駆動開発入門，ピアソン・エデュケーション (2003).
13) Holvaty, A. and Kaplan-Moss, J.: *Pro Django: Web Development Done Right*, Prentice Hall Open Software Development Series（2007 年 3 月 26 日発売予定）.

(平成 18 年 12 月 13 日受付)
(平成 19 年 3 月 12 日採録)

小菅 圭介（学生会員）

2006 年金沢工業大学大学院博士前期課程修了．同大学院博士後期課程在学中．工学修士．スクリプト言語 Python および Ruby，web アプリケーション，また Linux 各種ディストリビューションやメディアツールを用いたイラスト作成等に興味を持つ．2005 年度情報処理学会北陸支部より優秀学生賞受賞．

佐藤 規男（正会員）

1972 年東京大学工学部計数工学科卒業後，主に NTT 研究所の実用化部門にて，大規模交換機ソフト記述言語の処理系開発に従事．1998 年夏転職し日本ルーセントテクノロジー（株）にて FTTH プロジェクトに従事．1999 年秋に金沢工業大学情報工学科教授．Dionea の開発を着想し院生と開発．ほか，布結びの 3 次元 CG シミュレーションソフト “りぼん”，中国語発音教育用 3 次元 CG ソフト “音姐（yinjie）” 開発．工学博士．ACM，電子情報通信学会，芸術科学会各会員．