Vol.2021-SE-207 No.1
2021/3/1

Webベースの対話的な文芸的デバッグ環境の試作

杉山 朔太郎[1,a)]　小林 隆志[1,b)]

概要：本研究では，既存のスクリプタブルデバッグを発展させ，再現性の高いバグ報告を可能とする対話的な文芸的デバッグ環境を提案する．提案手法はデバッガを操作する実行可能なスクリプト記述，その実行によりデバッガを通して得られる情報とその可視化形式，説明用のテキストを織り交ぜた文書の形で保存する機能を提供する．それらの文書を用いることで，プログラムの実行時の振る舞いの詳細な観測や着目した振る舞いが起こる状況を開発者間で共有することが可能となる．本稿では，提案する対話的な文芸的デバッグ環境を説明し，我々が Jupyter を用いた Web アプリケーションとして試作したツール JISDLab について紹介する．

キーワード：Literate Debugging, Scriptable Debugger, Jupyter, Dynamic Analysis, Software Maintenance

1. はじめに

開発者はデバッグの工程においてブレークポイントの設置やステップ実行などの個々のデバッグ作業またはそれらを組み合わせた一連の作業を繰り返し行うことが頻繁にある [1]. 同時に，それらの作業は開発者にとって時間的，精神的に大きな負担となりうる．

そのようなデバッグ作業に伴う負担を軽減する方法として，スクリプタブルデバッガ (以下 SD と表記) が存在する．SD は記述したスクリプトに従って動作するデバッガであり，デバッグ手順の自動化・再利用を可能とすることでデバッグ作業の負担軽減を図るものである．

SD の中には，コマンドラインで利用可能な対話的なデバッガである GDB [2] で使用されるようなコマンドをスクリプトで記述するもの (対話型 SD と呼ぶ) や変数値の変更・メソッドの入退出・行通過等のイベント発生時に行う処理をスクリプトで記述するもの (リアクティブ型 SD と呼ぶ)，デバッグ動作をそのままの順序で命令的に記述することに加え観測値への非同期アクセスが可能なもの (命令型 SD と呼ぶ) などが存在する．

これまでにさまざまな SD が提案されてきたが，それらの適用範囲やデバッグ動作の記述しやすさの面で得意とするデバッグシナリオはそれぞれ異なる．例えば，対話型 SD はある観測ポイントでの詳細なデバッグ情報を取得するスクリプトを記述することが容易である反面，現在の実行に大きな影響を与えてはならないプログラムに適用できない．また，リアクティブ型 SD はデバッグ対象の内部変化が即座に分かる反面，過去の観測結果の履歴を利用したい場合やユーザが処理のタイミングを指定したい場合のスクリプトを記述しにくい．命令型 SD に関してはこれら対話型 SD，リアクティブ型 SD が苦手とする場合のスクリプト記述を得意とする反面，ある観測ポイントでの詳細なデバッグ情報を取得するスクリプトを記述することができないという課題が存在する．

しかしながら，いずれの既存 SD も，デバッグスクリプトを共有することができるが，そのスクリプトの実行環境を用意する必要があり，実行結果やそのスクリプトを利用するための前提条件などの付帯情報を一括して管理することができない．

本研究の目的は，このような既存 SD を用いたデバッグの問題を解決する，文芸的デバッグ (Literate Debugging，以下 LD) を実現することである．文芸的デバッグとはスクリプタブルデバッグと文芸的プログラミング [3] を融合させ，スクリプトだけでなく状況・経緯を説明するテキストやデバッグ実行結果も同時に文書として保存することでデバッグ作業の共有を容易にし，SD の再利用性の促進を図るデバッグ方法である．

本研究では文芸的デバッグを実現するため，Web ベースの対話的なスクリプタブルデバッグ環境 (以下 SD 環境) と本環境で用いるデバッグ用 API を提案する．また，それらを Java 言語向けにそれぞれ JISDLab, JISD として実装する．JISDLab 及び JISD は既存の命令型 SD である ImperSD [4] を元にして実装されている．これによりデバッグ対象となるプログラムに与える影響を最小限にしたデバッグを行えるという利点を享受できる．

しかし，ImperSD [4] そのものには実用上の課題が存在する．この実用上の課題は ImperSD の持つ機能や設計に関する制約から生じるものである．ImperSD が持つ制約とはブレークポイントの設置によるデバッグをサポートしていない，短時間で終了するプログラムのデバッグができない，仮想ファイルシステムで実行時情報の観測結果を管理

[1] 東京工業大学 情報理工学院
School of Computing, Tokyo Institute of Technology
[a)] sugiyama@sa.cs.titech.ac.jp
[b)] tkobaya@c.titech.ac.jp

している等である．一方，それらそれぞれの制約から生じる実用上の課題とは主にプログラムの詳細な振る舞いを調査できない，調査可能なプログラムの範囲が狭い，ある振る舞いが観測された状況や経緯を共有しづらい等の課題である．これらの課題に対処すべく JISDLab 及び JISD は ImperSD に対話型 SD としての要素を追加し，対話型 SD と命令型 SD のそれぞれの欠点を相互補完する形で発展させ，さらに SD の再利用性を促進するために Web から対話的にアクセスできるように実装した．

本研究では提案手法 *JISDLab* の有用性を議論するために，Apache Tomcat を用いた Web アプリケーションの実行時のデバッグを想定して JISDLab を用いたスクリプトの記述を行い，既存の ImperSD との比較を行った．その結果，プログラムの詳細な振る舞いが調査可能になったことや調査可能なプログラムの範囲が拡大したこと，スクリプト・説明テキスト・実行結果の文書化や観測結果のグラフによる即時可視化に伴いデバッグ作業の共有がしやすくなることといった提案手法及びツールの利点を確認した．

2. 関連研究

2.1 文芸的プログラミング

文芸的プログラミング [3] は，Donald Knuth が提唱したコンピュータのプログラミングスタイルであり，フォーマットされた自然言語テキスト，実行可能なコードスニペット，計算結果を織り交ぜることでプログラムの理解を支援する．近年では，データサイエンス分野では，Jupyter Notebook [5] に代表される対話型の文芸的プログラミングツールが広く利用されている [6]．Jupyter はプログラミングのチュートリアルや講義ノートとして配布・共有されている．100 万以上の Jupyter Notebook を調査した最近の研究では，93%は Python を用いたものであるが Ruby, C++などほかの言語を用いるものも一定数存在おり，ノートブック内には説明用の Markdown セルが 25%を占めるという報告がなされている [7]．このように，コード内のコメントではなく，入力，実行結果，説明文書，グラフなどの付帯情報をカプセル化したデータとして共有することには広く活用され始めている．

2.2 スクリプタブルデバッガ (SD)

これまでにさまざまな SD が提案されているが，ここでは既存の SD を以下の 3 つのグループに分けるものとする．3 つのグループとは，対話型 SD，リアクティブ型 SD，命令型 SD のことを指す．

対話型 SD とは，デバッグコマンドをスクリプトで記述し，スクリプト実行後にブレークポイントへの到達や観測値等の情報が即時手元に返ってくるようなデバッガである．例えば，Duel [8]，Sindarin [9] 等がある．現在の GDB [2] や LLDB [10] ではコマンドをスクリプトとしてあらかじめ記述することができる．また，GDB/LLDB の C/C++，Python API が提供されており，この API を用いてスクリプトを記述することでデバッグの自動化を行うことができる．

リアクティブ型 SD とは，変数値の変更・メソッドの入退出・行通過等のイベント発生時に行う処理を記述するイベント指向のデバッガである．例えば，GDB を拡張した Dalek [1] や関数型リアクティブプログラミングを用いた FrTime で記述する MZTake [1]，SDN(Software-defined network) 向けの Scala で記述する SD である Simon [11]，タイムトラベルデバッグをサポートする Expositor [12] 等がある．

命令型 SD とは，デバッグ動作をそのままの順序で命令的にスクリプトを記述し，スクリプト実行結果が即時には返らないデバッガである．ユーザが後から観測値へ非同期にアクセスすることで実行中のプログラムへの影響を最小限に抑えることができる．現在，ImperSD [4] という命令型 SD が存在する．これは Java アプリケーションのデバッグ用に開発され，デバッグスクリプトを Java の REPL 環境 JShell 上で記述するものである．ユーザは MewdFS [13] というデバッグインターフェースが提供する API を用いて，ProbeJ [14] という値観測用 Java エージェントと Socket 通信を行い，観測ポイントの設定・破棄や観測結果の取得を行う．MewdFS は取得した観測情報を Windows の仮想ファイルシステム DokanFuse [15] 上にファイルとして格納すると同時に，MewdFS がファイルアクセス用 API を提供することでスクリプトでの観測結果の取得を容易にしている．一方で，ImperSD は通常のデバッガの有用な機能であるブレークポイントによるデバッグやステップ実行・スタックトレースの表示などの機能をサポートしていない．また，プログラム開始前に観測ポイントを設置できないため，すぐに終了してしまうプログラムのデバッグやプログラムの初めに 1 度だけ通過するような行への観測ポイントの設置を行うことができない．

3. 文芸的デバッグ環境

3.1 LD の要件

文芸的デバッグ (LD) は既存 SD が持つ共有性に関する課題を解決するためのデバッグ方法である．共有性に関する課題とは既存 SD ではプログラムのある振る舞いが観測された状況や経緯を共有しづらいことである．これは既存 SD がスクリプトとその実行結果やスクリプトを利用するための前提条件などの付帯情報を一括して管理できないことに起因する．それを踏まえ，LD の要件として「実行可能なデバッグスクリプトやその説明テキスト，実行結果・グラフといった一連の作業記録を同一箇所に保存可能であること」を定める．このようにスクリプトの付帯情報をカプセル化することでデバッグ作業の共有性の向上を図る．また，共有性の向上を考える上で，保存された記録へのアクセスの容易さも問題となる．そこで「さまざまな環境か

らデバッグ環境へのアクセスが可能であること」を LD のもう一つの要件として定める．

3.2 既存 SD 環境: ImperSD の特性

本環境は ImperSD の便利な機能や利点をなるべく保持しながら，同時に ImperSD に存在するいくつかの欠点を改善するように設計されている．ImperSD の主な利点と欠点それぞれについて以下に列挙する．

利点

- 観測結果に非同期にアクセスできる
- 同一箇所の複数の観測結果を保持する
- デバッグ時に対象プログラムに与える影響を最小限にできる
- 実行中のアプリケーションをデバッグできる
- プログラム実行前に静的情報を取得できる
- 外部プログラムと容易に連携できる

欠点

- 対象プログラムの詳細な実行時情報を取得できない
- 短時間で終了するプログラム等のデバッグができない
- ある振る舞いが観測された状況や経緯を共有しづらい

ImperSD の 1 つ目の欠点である対象プログラムの詳細な実行時情報を取得できないことは ImperSD がブレークポイントを設定する機能をサポートしていないことに起因する．これは ImperSD の構成要素のうち値観測を行う部分である ProbeJ がブレークポイントを設置するための権限を意図的に付与していないことに端を発する．同時にこの権限喪失により可能となる最適化が存在することによって ProbeJ が値観測時に対象プログラムに与える影響の小ささを保っていることが報告されている．一方，ブレークポイントを設置することによって可能となるある地点で観測可能な値情報やスタックトレースなどの詳細な情報はバグの原因を多角的に調査する際に大変有用である．ImperSD の 2 つ目の欠点である短時間で終了するプログラム等のデバッグができないことはツールの適用可能範囲を狭め，汎用的な SD としては価値が下がるため対処すべきである．

ImperSD の 3 つ目の欠点はある振る舞いが観測された状況や経緯を共有しづらいことである．これは観測結果の格納方法として仮想ファイルシステムを使用していることに起因する．仮想ファイルシステムにはプログラム構造と観測値のみが格納されており，どのような状況でその値が観測されたかの情報を提供しない．これはデバッグ作業の共有しやすさに関する問題となり，さらにデバッグ作業の再現性を脅かす可能性がある．

3.3 提供する機能

ここでは LD の要件を満たすとともに本環境の元となる ImperSD が持つ利点を保持しながらその欠点を改善するために，新たに提案する SD 環境が持つ機能について定義する．本環境が提供する主な機能は以下の通りである．

(1) ブレークポイントによるデバッグ

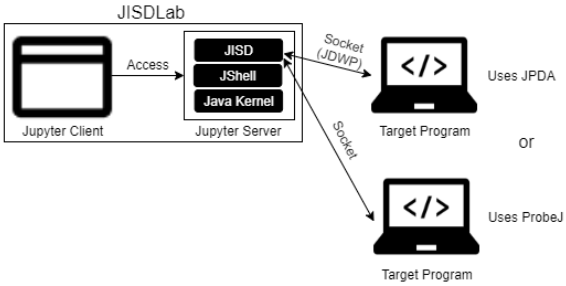

図 1 JISDLab の構成

(2) ステップ実行，メソッド実行，スタックトレースの表示等のデバッグ操作の提供
(3) Web からのスクリプトによる遠隔デバッグ
(4) Text，HTML/Markdown 形式での文書の挿入
(5) グラフの描画・保存
(6) 観測結果への非同期アクセス
(7) 同一箇所の複数の観測結果の保持
(8) 対象プログラムに与える影響を最小限にしたデバッグ
(9) 静的情報の提供
(10)外部プログラムとの容易な連携

前述の ImperSD の欠点に対処するために対話的 SD としての機能を新しく追加した (項目 1〜2)．また，LD の要件を満たすため，さまざまな環境からのデバッグや文書記述・グラフ描画等の文芸的プログラミング [3] の要素を追加した (項目 3〜5)．残りの項目 6〜10 は ImperSD の機能を引き継ぐために必要な機能である．

3.4 構成要素

本環境は以下の要素から構成される．

- Web サーバ
- Web クライアント
- 対象プログラム用の実行環境
- スクリプタブルデバッグ用 API/モニタリングツール
- SD の対話的な実行環境

本環境は Web を活用しさまざまな環境からのデバッグをサポートする．なお，SD の利点を促進する機能の提供のため，Web クライアントには文書の編集・保存，グラフ表示，コンソール連携等の機能が必要である．また，SD が対話的な実行環境で動作することにより逐次的なスクリプトの実行が可能であり，デバッグの際に頻発するバグ原因に関する仮説検証を容易に行うことができる．モニタリングツールは命令型 SD の機能を実現するために必要であり，なるべくプログラムに観測の影響を与えないようにするべきである．

4. 実装: *JISDLab*

本研究で提示した提案手法を Java アプリケーションを対象とした SD 環境 JISDLab とその構成要素であるデバッグ用 API JISD として実装した．以下，それぞれについて説

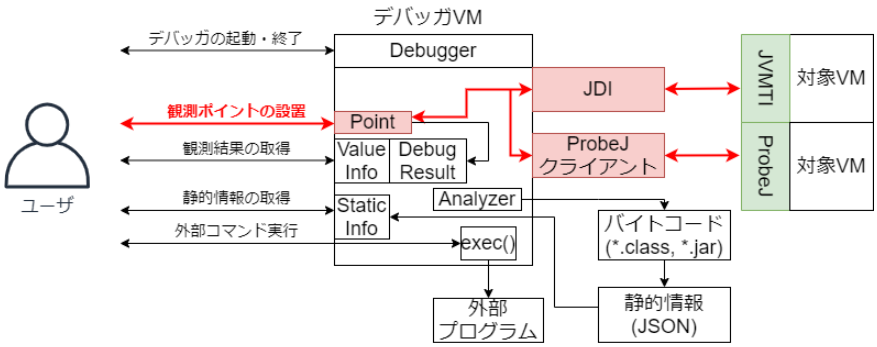

図 2 JISD の構成

明する．

4.1 JISDLab

JISDLab の構成を図 1 に示す．JISDLab はスクリプト用言語として Java を採用している．これにより Java アプリケーションの開発者が実装言語と同じ言語で容易にデバッグスクリプトを記述することができる．なお，スクリプト言語の想定する JDK のバージョンは 11 である．

Web 環境と SD 用の対話的な実行環境には JupyterLab [16] を採用した．JupyterLab はコード実行やテキスト文書の挿入，グラフの表示・保存，コンソールとの接続等の機能を有する文芸的プログラミングのためのシステムである．Jupyter クライアントとして利用可能な Web ブラウザや IDE 等さまざまな環境でデバッグを行えるという利点を享受することが可能になっている．また，Java 用 Jupyter カーネルとして OSS として Github 上に公開されている IJava [17] をフォークして用いた．IJava は Java9 から導入された REPL 環境である JShell を内部で用いており，Jupyter Notebook において Java をサポートする．また，Maven リポジトリからのライブラリのインストールやコンソールでのコマンド実行等をマジックコマンドとして提供している．

4.2 JISD

JISD の構造の概略図を図 2 に示す．JISDLab のコアな構成要素である JISD はデバッグ用 API を提供するライブラリである．JISD は観測ポイントの設置や取得，静的情報の取得，外部コマンドの実行等の機能を有している．対話型 SD としての機能であるブレークポイントの設定機能や，命令型 SD としての機能である観測結果の非同期アクセスを可能にするための観測ポイント (プルーブポイントと呼称) を実装している．JISD は JISDLab 内において Jupyter カーネル上の jdk.jshell インターフェースを通して使用されるが，JISD 自体は jar で提供される単なる Java ライブラリのため JISDLab 以外の Java 環境でも使用できる．

対話型 SD を実現するために Java Platform Debugger Architecture(JPDA) [18] という Java アプリケーションのデバッグのための仕組みを利用している．JPDA は JVMTI, JDWP，JDI の 3 層からなる．JVMTI は実行中の Java プログラムの内部状態観測・実行制御を行うインタフェースであり，JDWP は対象プログラムとデバッガのプロセス間で行われる通信プロトコルである．JDI はデバッグアプリケーションを容易に記述することができるインターフェースである．JISD はこれらを用いることでブレークポイントの設定やステップ実行，観測値の取得等，基本的なデバッグ操作を提供する．また，ブレーク中にメソッド実行を行うことができその返り値を取得することもできる．これらの機能を用いることで，ユーザはプログラムの詳細な振る舞いを調査することができる．

また，命令型 SD を実現するために ProbeJ [14] を利用している．ProbeJ は前述の JPDA のうち JVMTI を用いて Java アプリケーションから値の観測を行うモニタリングツールであり，Java のエージェントとして実装されている．あらかじめ設定された観測ポイントに到達すると値の観測のみを行い，観測後はプログラムを再開する．観測値は一定数保持され，後から Socket 通信を用いて観測値に問い合わせることができる．これによりプログラムに与える影響を最小限にして変数のモニタリングをすることが可能になっている．JISD はこの ProbeJ と通信することにより実行中プログラムに最小の影響で観測値や観測時の時刻等のデバッグ情報を得ることができる．

観測結果は DebugResult クラスである行の変数ごとに管理する．DebugResult は for 文による反復等で同じ行の変数の値を複数観測する状況に対応するため，個々の値を ValueInfo クラスで管理する．DebugResult クラスは ValueInfo インスタンスを保持する上限を定め，それらをキューで管理する．そのため，常に最新の値を保持し，保持上限に達した場合は過去に観測されたもののうち古い方から削除される．また，ValueInfo は配列，参照型変数に対応するため，それぞれ ArrayInfo，ObjectInfo として ValueInfo を継承したクラスを定義した．観測結果作成時にこれらを検出した場合，あらかじめ決められた回数だけ配列の要素や参照型変数のフィールドを展開して値を保存する．この展開回数は DebugResult によって定められる．ただし，プリミティブ型変数は PrimitiveInfo として定義され，これに関し

ては要素等の子要素を持たないため，展開は行われない．この展開機能によってさまざまな種類の変数が保存可能になり，後からそれらの観測結果を再利用可能になる．

静的情報の提供時には，まず対象プログラムの実行ファイル群をバイトコード解析して抽出し，メタデータとして json ファイルに出力する．このデータはユーザが必要とするときに StaticInfo インスタンスを作成し，そのメソッドを通してアクセスされる．静的情報として提供されるものは対象プログラムのソースコードやクラス，メソッド，フィールド，ローカル変数，観測ポイントが設置可能な行番号等である．これにより，ユーザは事前に静的情報を把握した上で観測ポイントの設置等のデバッグ操作をすることが可能になる．

外部プログラムとの連携のために exec メソッドが実装されている．exec メソッドは引数に与えられた文字列をシェルコマンドとして実行する静的メソッドであり，実行後標準出力/エラー出力の内容，終了コードを返す．これにより，コマンド実行結果の再利用が可能である．

4.3 使用方法

ここでは JISDLab の使用方法について説明する．JISDLab は Docker による環境構築をサポートしており，あらゆる環境において少ないコマンドで容易にインストールできる．ユーザは Web ブラウザや Jupyter 対応の IDE 等から JISDLab にアクセスできる．アクセス後，JISDLab の中で JISD ライブラリを用いながらスクリプタブルデバッグを行う．JISD によるデバッグの大まかな流れを以下で説明する．JISD の Javadoc は `https://sakupo.github.io/JISD-doc/` から参照できる．ユーザはまず，jisd.debug.Debugger クラスのインスタンスを作成し，デバッガを定義する．その際，どのプログラムをデバッグするかやそのデバッガが対話型か命令型か等を引数で指定する．次に，ユーザは Debugger.run メソッドでそのデバッガを起動する．このとき，デバッガ起動後何秒待機するかを引数で指定することもできる．これにより，観測ポイントの設置等その後に続くデバッグ動作のタイミング指定が可能である．

観測ポイントは Debugger.stopAt または Debugger.watch メソッドを呼び出すことで設定でき，この返り値として Optional な Point 型のインスタンスが返る．観測ポイント到達時，stopAt の場合は対象プログラムを停止したままにするが，watch メソッドの場合値観測後直ちに対象プログラムを再開する．Point クラスは観測ポイントを管理するクラスであり，観測結果の取り出し (getResult) や観測ポイントの削除 (clear) 等が可能である．

Point.getResult メソッドで取り出された観測結果は DebugResult インスタンスとして保存されている．DebugResult クラスは観測された個々の値情報を管理する複数の ValueInfo インスタンスをキューで管理する．なお，DebugResult クラスの保持する観測結果 (ValueInfo) の個数の上限は変

App.java

```
package tomcatdemo;

import ...

@WebServlet("/home")
public class App extends HttpServlet {
  static int noOfViews = 0;

  @Override
  protected void doGet(HttpServletRequest req, HttpServletResponse res) throws IOException {
    /* start */
    /* section 1 */
    sleep( seconds: 0.3);
    /* section 2 */
    sleep( seconds: 0.1);
    /* section 3 */
    sleep( seconds: 5.0);
    /* section 4 */
    sleep( seconds: 0.2);
    /* end */
    res.setContentType("text/html; charset=utf-8");
    String message = "Hello World!";
    try (PrintWriter out = res.getWriter()) {
      out.println("<html><body>");
      out.println(message + "<br>");
      out.println("views: " + noOfViews++);
      out.println("</body></html>");
    }
  }

  void sleep(double seconds) {...}
}
```

図 3 対象 Tomcat アプリのコード

更することができる．ブレークポイントで停止中に限り，ValueInfo クラスが観測した値が配列，参照型の場合はそれが持つ要素，フィールドの値を展開して参照することが可能である．展開後の値はブレークポイントからの復帰後も参照が可能となる．また，観測値が配列，参照型の時に自動で展開する回数を変数ごとにあらかじめ設定することもできる．

5. 評価

5.1 目的

本実験の目的を 2 つ述べる．第一の目的は提案手法が 1 節で述べたような，既存 SD を用いたデバッグに関する問題を解決する文芸的デバッグ (Literate Debugging) のための環境を実現できているか示すことである．また，第二の目的は JISDLab が現状の ImperSD の課題に対処し，SD としての実用性を向上させるものであるか示すことである．

5.2 方法

本節では起動中の Tomcat アプリのデバッグ操作を例として JISDLab を ImperSD と比較しながら評価する．Apache Tomcat [19] は Java で実装された Web コンテナであり，HTTP サーバを内包しているため単体でサーバとして機能する．本研究では JISDLab の対話型 SD としての側面と命令型 SD としての側面の 2 つの側面を評価するため，それぞれの場合で利用ケースを想定した．Case1: HTTP リクエストの中身が知りたい場合，Case2: ページ表示のボトルネックとなる部分を知りたい場合の 2 つのケースを想定し，前者は対話型の機能，後者は命令型の機能を使用して評価する．なお，ImperSD は接続時の静的解析用に jar ファイルを指定することができない等の機能不十分により実行中の Tomcat アプリに接続することができない．そのため，接続できたことを仮定してデバッグスクリプトを記述するものとする．ただし，ImperSD には対話型の機能が備わっていないため命令型の機能を使用する Case2 のみ実施する．

図 4 JISDLab を用いた Case1 のデバッグ作業例

```
1  var points = new ArrayList<Probe>();
2  int[] lines = {18, 20, 22, 24, 26};
3  Arrays.stream(lines).forEach(line -> {
4    // 観測ポイントの設定
5    points.add(new Probe("y:/case2/tomcatdemo/App/doGet/noOfViews/"+line));
6  })
7  // ここで http://localhost:8080/TomcatDemo-0.0.1/home に1回アクセス
8  points.forEach(p -> {
9    var values = p.getValsList();
10   var times = p.getTimeValsList();
11   System.out.println("value\t:\ttimestamp");
12   for (int i = 0; i < values.size(); i++) {
13     System.out.println(values.get(i)+"\t:\t"+times.get(i));
14   }
15 })
```

図 5 ImperSD を用いた Case2 のデバッグスクリプト

5.2.1 データセット

2つのケース共通で用いる対象のソースコード App.java を図 3 に示す．このコードは Web 上から/home に GET メソッドで HTTP リクエストが送られてきた時に現在までの/home のページ閲覧数を HTTP レスポンスとして返すものである．また, start コメントと end コメントに挟まれた部分は section1 から section4 の 4 つのセクションに分けており，実行時間のかかる何らかの重い処理を sleep メソッドを使用して擬似的に表現している．sleep メソッドは Java 標準の Thread.sleep メソッドをラップするメソッドであり，引数として現スレッドを停止する秒数を double 型で受け取るものである．本研究では JISDLab はこの Tomcat アプリと同一のマシン上のブラウザから Jupyter サーバに接続し，JISD の API を通して Tomcat アプリのデバッグポートに接続することでデバッグを試みるものとする．また，Tomcat サーバは http://localhost:8080/TomcatDemo-0.0.1/home へのアクセスを前述の/home にルーティングするものとする．

5.2.2 手順

想定する 2 つの各ケースについて実験手順を説明する．2 つのケースとも以下の手順に従う．1. JISDLab, ImperSD の各ツールを個別に使用してデバッグスクリプトの記述を試みる 2. スクリプトを実行し，デバッガを起動した後，前述の App.java コード内に観測ポイントを設定する．3. ブラウザから 1 度 http://localhost:8080/TomcatDemo-0.0.1/home にアクセスする．4. スクリプトを実行し，観測結果を受け取る．5. Case2 では観測結果をグラフとして出力する．

5.2.3 評価基準

Case1 では HTTP リクエストの詳細情報を観測できたならば，観測に使用した SD がプログラムの詳細な振る舞いを観測する能力があると評価する．Case2 では対象の Tomcat WebApp に含まれる 4 つのセクションのうち，実行時間の

最も多くかかる部分を特定できたか，そうでないかを評価する．また，スクリプトの記述しやすさや観測結果の理解しやすさ，一連のデバッグ作業をその状況を含めての共有しやすさについても評価の考慮に入れるものとする．

5.3 Case1: 対話型 SD としての側面

HTTP リクエストの中身が知りたい場合の JISDLab を用いたデバッグ作業例を図 4 に示す．このシナリオを実行するためにあらかじめ localhost の 8080/tcp ポート番号で HTTP リクエストを待機するような Tomcat サーバを起動させている．このとき，JPDA によるデバッグが行うため 25432/tcp ポートでデバッガ接続を待ち受けるように起動時のオプションを設定している．まず，1 番目のセルにてデバッガを定義する．Debugger のコンストラクタの引数にデバッグ対象へのアドレスとポートを指定する．次に，2 番目のセルの 1 行目でデバッガを起動し，同セルの 2 行目で tomcatdemo.App クラスの 18 行目にブレークポイントの設定を行なっている．ここで，Markdown セルの記述通りに `http://localhost:8080/TomcatDemo-0.0.1/home` に Google Chrome ブラウザから 1 回アクセスし，ブレークポイントへの到達を待機する．この際，ユーザは所定の URL にアクセスするのにリンクをクリックするだけで良い．その後，3 番目のセルで変数 req の観測値を getResults メソッドによって取得し，getLatestValue メソッドによって最新の値を抽出している．4 番目以降のセルでは取得した変数 req の持つメソッドを invokeMethod メソッドで対象 VM 内で実行させ返り値を確認することで，デバッグに必要な情報を取得している．すなわち，4，5，6 番目のセルではそれぞれ HTTP メソッド，リクエスト URI，User-Agent ヘッダを取得するものである．

5.4 Case2: 命令型 SD としての側面

Case2 では，localhost の 8080/tcp ポートで HTTP リクエストを待機し，39876/tcp ポートで ProbeJ のデバッガ接続を待ち受けるように起動時のオプションを設定して Tomcat サーバを起動させていると仮定する．この状態で，対象 Tomcat アプリでのページ表示のボトルネックとなる部分を調査することを想定する．

まず，ImperSD を用いたデバッグ作業例について述べる．ImperSD は観測結果にアクセスするため，コマンドライン上で ImperSD が提供する mount コマンドで ProbeJ と通信する仮想ファイルシステムを `Y:/case2` に作成するとともにデバッグ対象プログラムと接続する．次に script コマンドで JShell を起動させ，その中で図 5 のスクリプトを実行する．7 行目以降は結果の受け取りを行い，値とタイムスタンプを表示している．JISDLab の場合と比べ観測ポイントの指定方法や管理方法等細かな差はあるがスクリプトの記述内容に大差はない．グラフ出力には別途何らかの描画プログラムを呼び出す必要があるが本論文では省略する．

JISDLab を用いた場合のデバッグ作業例を図 6 に示す．

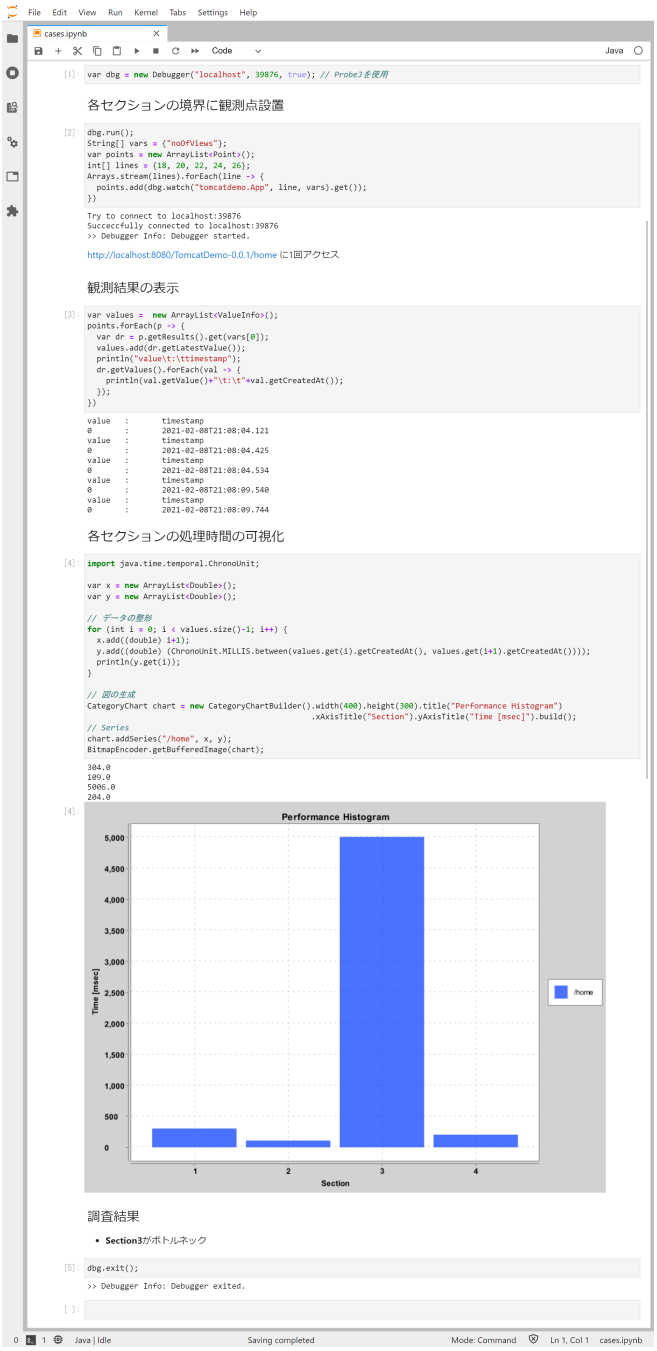

図 6 JISDLab を用いた Case2 のデバッグ作業例

まず, Case1 と同様に 1 番目のセルにてデバッガを定義する．次に，2 番目のセルにてデバッガ起動後 tomcatdemo.App クラス内で定義される変数 noOfViews について 18, 20, 22, 24, 26 行の 5 つの行で値を観測するように watch メソッドを用いている．なお, 観測値は後から参照できるように観測ポイントを扱う Point 型の ArrayList である points に 5 つの観測ポイントを格納している．ここで，Markdown セルの記述通りに `http://localhost:8080/TomcatDemo-0.0.1/home` に Google Chrome ブラウザから 1 回アクセスし，観測ポイントへの到達を待機する．この後，3 番目のセルにて各観測ポイントから観測情報を取得し，それぞれについて値とタイムスタンプを表示させている．タイムスタンプは値が観

測された瞬間の日時を表す．

また，これ例では，各観測ポイントでの観測時刻の差から tomcatdemo.App クラス内の各セクションの実行時間を計算し，グラフとして描画するためのスクリプトを 4 番目のセルに記述しており，そのスクリプトを実行した時の出力が表示されている．グラフ出力例より，セクション 3 の実行時間が長いことが分かり/home において同セクションがボトルネックとなっていることが視覚的に確認できる．また，報告文書により明確にその調査結果を残している．

このファイルを保管することで，デバッグ作業のためのスクリプトだけでなく，その実行結果と可視化表現を容易に共有でき，JISDLab は本研究の目的としていた LD を実現する環境として十分な機能を有することが確認できた．

5.5 考察

Case1 と Case2 の 2 つの利用例および ImperSD でのデバッグ作業例との比較から，JISDLab は ImperSD を以下の方法で改善したといえる．つまり，ブレークポイントを用いたデバッグのサポートによるツールの適用範囲拡大，グラフ描画機能を用いた観測結果の可視化による視覚的なプログラム理解への支援，文芸的デバッグによるデバッグ作業の文書化，共有，再現の支援である．また，JISDLab は観測結果への非同期アクセスや静的情報の提供，外部プログラムとの容易な連携等 ImperSD の提供する機能をほぼ全て保持しており，JISDLab は ImperSD の上位互換の SD 環境であると言える．

6. おわりに

本研究では既存 SD の共有性に関する課題を解決する文芸的デバッグを実現するため，Web ベースの対話的な SD 環境を提案した．また，その提案手法を実現するものとして Java 向けの SD 環境 JISDLab とデバッグ用 API JISD を既存の命令型 SD である ImperSD を改良する形で実装した．

提案手法の有用性を議論するために，Apache Tomcat を用いた Web アプリケーションの実行時のデバッグを想定して JISDLab を用いたスクリプトの記述を行い，JISDLab と ImperSD の比較を通して評価を行った．その結果, JISDLab は ImperSD をその機能を保持しながらさまざまな面で改善する SD 環境であることを示した．

今後の展望として，JISDLab を用いた被験者実験の実施・評価，さらなるデバッグシナリオの検討・評価，JISDLab のパフォーマンス評価等が考えられる．また，現状 JISDLab は命令型 SD 機能が Windows のみのサポートに留まっており，観測できる変数の型に制約があることや一部の静的情報しか提供しないこと等ツールとして改善の余地が存在する．このような制約を改善し，機能強化を行うことでさらなる適用範囲の拡大や使用性の向上が期待できる．

謝辞 本研究の一部は科研費（#18H03221）の助成を受けた．

参考文献

[1] G. Marceau, G. H. Cooper, J. P. Spiro, S. Krishnamurthi, and S. P. Reiss. The design and implementation of a dataflow language for scriptable debugging. *Automated Software Engineering*, Vol. 14, No. 1, pp. 59–86, 2007.
[2] GDB, 2021/1/23. https://www.gnu.org/software/gdb/.
[3] D. E. Knuth. Literate Programming. *The Computer Journal*, Vol. 27, No. 2, pp. 97–111, 01 1984.
[4] 平ノ内, 小林. ImperSD: Java 言語向け命令型スクリプタブルデバッグ環境. 信学技報 SS2020-3, 電子情報通信学会, July 2020.
[5] F. Pérez and B. E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, Vol. 9, No. 3, pp. 21–29, 2007.
[6] M. B. Kery, et al. The story in the notebook: Exploratory data science using a literate programming tool. In *Proc. CHI*, p. 1–11, 2018.
[7] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire. A large-scale study about quality and reproducibility of jupyter notebooks. In *Proc. MSR*, pp. 507–517, 2019.
[8] M. Golan and D. R. Hanson. Duel: a very high-level debugging language. In *USENIX Winter*, Vol. 107, p. 118. Citeseer, 1993.
[9] T. Dupriez, G. Polito, S. Costiou, V. Aranega, and S. Ducasse. Sindarin: A versatile scripting api for the pharo debugger. In *Proc. DLS*, p. 67–79, 2019.
[10] LLDB, 2021/1/23. https://lldb.llvm.org/.
[11] T. Nelson, D. Yu, Y. Li, R. Fonseca, and S. Krishnamurthi. Simon: Scriptable interactive monitoring for SDNs. In *in Proc. SOSR*, p. 19, 2015.
[12] Y. P. Khoo, J. S. Foster, and M. Hicks. Expositor: scriptable time-travel debugging with first-class traces. In *Proc. ICSE*, pp. 352–361, 2013.
[13] 平ノ内, 野田, 小林. 仮想ファイルシステムを用いたプログラム内部状態観測ツールの試作. 信学技報 SS2018-13, 電子情報通信学会, July 2018.
[14] 嶋利, 石尾, 井上. ソフトウェアの実行を分析するための低侵襲なモニタリングツールの試作. ソフトウェアエンジニアリングシンポジウム 2017 論文集, pp. 224–227, 2017.
[15] Dokan FUSE, 2021/1/23. https://github.com/dokan-dev/dokany.
[16] JupyterLab, 2021/1/23. https://jupyterlab.readthedocs.io/en/stable/.
[17] IJava, 2021/1/23. https://github.com/sakupo/IJava.
[18] JPDA, 2021/1/23. https://docs.oracle.com/javase/jp/1.5.0/guide/jpda/architecture.html.
[19] Apache Tomcat, 2021/1/23. http://tomcat.apache.org/.