

**Rational™ PurifyPlus**  
**Rational™ Purify®**  
**Rational™ PureCoverage**  
**Rational™ Quantify®**

ファースト ステップ

バージョン: 2003.06.10

GI10-3260-00

WINDOWS



## ご注意

### 著作権の表示

Copyright © 2001-2003, Rational Software Corporation. All rights reserved.

バージョン番号: 2003.06.10

### 許可される使用

本書には、Rational Software Corporation (または「Rational」) が所有権を有する固有情報が含まれており、個人を対象に、Rational 製品の操作、メンテナンスに関する情報を提供しています。本書の一部またはその全部を他の目的に使用することは禁止されており、形態あるいは方法を問わず、Rational の事前の許可を得ないで、複製、コピー、改変、開示、配布、送信、検索可能なシステムへの保存、人間またはコンピュータの言語への翻訳を行うことはできません。

### 商標

Rational、Rational Software Corporation、ClearQuest、PureCoverage、Purify、Purify'd、Quantify、および Rational Visual Test は、Rational Software Corporation の米国およびその他の国における商標または登録商標です。その他の名前はすべて、識別の目的でのみ使用されているものであり、それぞれの会社の商標または登録商標です。

Microsoft、Virtual Basic、Visual C++、Visual Studio、および Windows は、Microsoft Corporation の米国およびその他の国における商標または登録商標です。

### 特許

U.S. 特許番号 5,193,180、5,335,344、5,535,329、5,835,701。このほかにも特許申請中。

Purify は、Sun Microsystems, Inc. の U.S. 特許番号 5,404,499 の下にライセンス供与されています。

### 米国政府の権利

米国政府による使用、複製、または開示は、該当する Rational Software Corporation ライセンス契約書および DFARS 277.7202-1(a) および 277.7202-3(a) (1995)、DFARS 252.227-7013(c)(1)(ii) (Oct. 1988)、FAR 12.212(a) (1995)、FAR 52.227-19、または FAR 227-14 で定められている規定の制約を受けます。

### 免責事項

本書および関連ソフトウェアは、ライセンス契約に基づいて使用することができます。ライセンス契約で明示的に定められていない限り、Rational Software Corporation は、本メディア、ソフトウェア製品、およびその関連文書について、明示的にも暗黙的にも、商品性に関する保証、特定目的への適合性に関する保証、取り扱い、使用、または取引行為に伴う保証について一切の責任を負いません。



# 目次

<b>Rational PurifyPlus 製品ファミリーの概要</b>	<b>1</b>
Rational PurifyPlus: 概要	1
開発者のためのヒント	2
品質管理エンジニアのためのヒント	3
PurifyPlus のその他の機能について	4
Rational 技術サポートの連絡先	5
<b>Rational Purify ファースト ステップ</b>	<b>7</b>
Visual C/C++ の開発者とテスターの皆さんへ	7
Purify と Visual C/C++: 主な機能	7
Purify と Visual C/C++: 基本手順	9
Purify と Visual C/C++: 高度な機能	21
Java の開発者とテスターの皆さんへ	27
Purify と Java: 主な機能	27
Purify と Java: 基本手順	29
Purify と Java: 高度な機能	38
.NET 管理コードの開発者とテスターの皆さんへ	42
Purify と .NET 管理コード: 主な機能	42
Purify と .NET 管理コード: 基本手順	43
Purify と .NET 管理コード: 高度な機能	52
<b>Rational PureCoverage ファースト ステップ</b>	<b>57</b>
PureCoverage: 主な機能	57
PureCoverage: 基本手順	59
PureCoverage: 高度な機能	66
<b>Rational Quantify ファースト ステップ</b>	<b>73</b>
Quantify: 主な機能	73
Quantify: 基本手順	74
Quantify: 高度な機能	85
<b>索引</b>	<b>95</b>



# Rational PurifyPlus 製品 ファミリーの概要

## Rational PurifyPlus: 概要

---

Rational™ PurifyPlus (以下 PurifyPlus) は、高品質なアプリケーションをより効率的に開発するために不可欠な次の 3 つのツールを統合したものです。

- **Rational Purify®** (以下 Purify): プログラムのすべてのコンポーネントでランタイムエラーとメモリリークを検出する、自動エラー検出ツール
- **Rational Quantify®** (以下 Quantify): パフォーマンスのボトルネックを解決することによって、プログラムの実行速度の向上を図る、パフォーマンス分析ツール
- **Rational PureCoverage** (以下 PureCoverage): リリース前にコードがすべてテストされていることを確認する、コードカバレッジツール

これらの 3 つのツールは操作が簡単な一方で、Visual C/C++、Visual Basic、Java、または Microsoft Visual Studio .NET が対応する言語で作成された管理コードを使用して、高パフォーマンスのより安定したアプリケーションを開発するための貴重なデータを提供します。

Microsoft Visual Studio でコードを開発する場合は、Visual Studio のメニューまたはツールバーから PurifyPlus の各ツールを起動します。たとえば、Visual Studio のデバッガとエディタを Purify と共に使用すると、ソフトウェアに潜む障害を短時間で修正できます。また、Visual Studio のリソースの一部のみが必要な場合は、各ツールを単独使用アプリケーションとして使用することも可能です。

ソフトウェアをテストする場合は、PurifyPlus の各ツールを既存のテストスクリプトやルーチンで使用すると、エラー検出、メモリプロファイリング、コードカバレッジモニター、パフォーマンス測定を自動化できます。開発の初期段階から夜間テストで各ツールを使用して、リグレッションなどの問題を発生と同時に検出できるようにします。

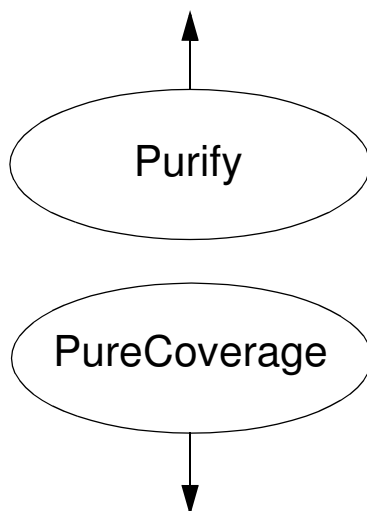
役立つツールを使用しない手はありません。PurifyPlus では数秒で問題を特定できるので、問題の検出に時間をかける必要がなくなります。また、障害が簡単に検出できるので、障害を残したまま製品をリリースしてしまうようなこともなくなります。PurifyPlus の各ツールを開発早期から出荷時まで継続して使用することで、開発側も顧客側も多大なメリットを得ることができます。

## 開発者のためのヒント

ここでは、PurifyPlus を使用してより高パフォーマンスの安定したコードを開発するためのヒントを示します。

### メモリ エラーの早期検出

コーディングしながら Purify を使用すると、検出が困難な障害を特定できます。メモリ エラーは、エラーの症状がすぐに現れるとは限りませんが、将来的にプログラムをクラッシュさせるものです。



### コード カバレッジの向上

実行していないコードは Purify 化されていません。

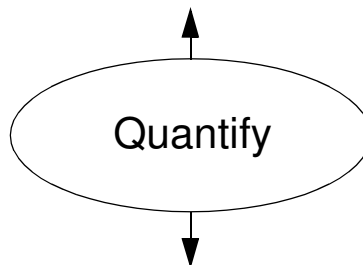
PureCoverage を使用して、チェックイン前のテスト時にすべてのコードがテストされたかどうかをチェックします。

C/C++ コードの場合は、Purify 内から PureCoverage を起動できます。これには、Purify の [プログラムの実行] ダイアログ ボックスの [カバレッジ、エラー、リーク データ] をクリックします。

### パフォーマンスのボトルネックの回避

新しいコードを作成したり既存のコードを修正したりするときは、パフォーマンス ボトルネックになる前に Quantify で継続的にパフォーマンスが低下している箇所を検出します。

Quantify を使用すると、より効率的なコードを作成するための情報を得ることができます。チームの全員をパフォーマンス エンジニアにすることも可能です。



### コード構造の分析

新しいコードを作成する一般的な理由は、プログラムのパフォーマンスを向上させることにあります。しかし、コードは何年にもわたっていろいろな開発者の手により開発されているのが普通です。このようなコードのパフォーマンスを効率的に向上させるにはどのようにしたらよいでしょうか。

Quantify では、パフォーマンスのボトルネックを検出できるだけでなくコード構造を分析することもできます。コード構造を分析することは、パフォーマンスを効率的に向上させるのに役立ちます。

## 品質管理エンジニアのためのヒント

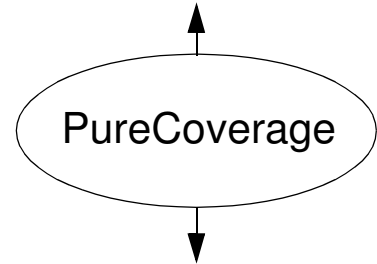
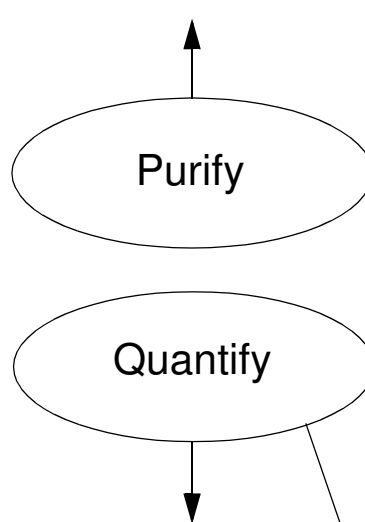
ここでは、PurifyPlus を使用してソフトウェアの品質を維持するためのヒントを示します。

### コードの内部エラーの検出

最適な結果を得るには、Purify 化したバージョンのプログラムに対してすべてのテストを実行します。これにより、外部機能テストでは発見できない内部メモリの問題が検出されます。

### すべてのコードの日次テスト

すべてのコードをテストしていることを確認するには、PureCoverage を毎日継続的に使用します。プログラムを継続的にモニターすることにより、追加した修正変更がすべてテストされているかどうかを確認することも可能です。



### パフォーマンスが向上した場合

パフォーマンスが予想以上に向上した場合、あるセクションのコードがテストされていない疑いがあります。最後の PureCoverage の結果と以前のランを比較して、同レベルのカバレッジが報告されているかどうかを調べます。

### コード カバレッジが低下した場合

コード カバレッジが低下した場合、既存のテストが新しいコードをテストしていないことが考えられます。または、新しいコードに障害があるため、あるセクションのコードがテストされていない可能性もあります。新しいコードをテストするテスト ケースを作成するには、Rational Robot (以下 Robot)、Rational Visual Test (以下 Visual Test) などの自動テスト ツールを使用します。

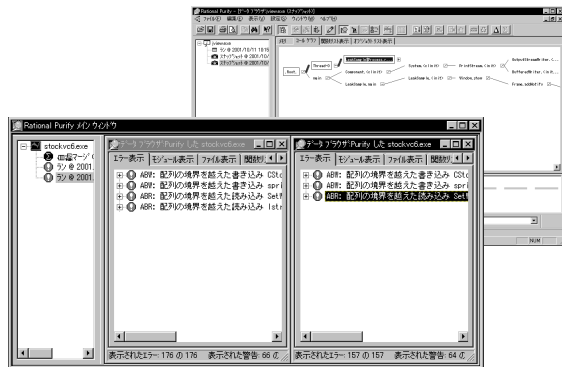
### パフォーマンスが低下した場合

パフォーマンスが突然低下した場合は、最後にチェックインしたコードが原因であると考えられます。Quantify を使用して、パフォーマンスが許容範囲内だった以前のランの結果と比較してみると、速度が低下したプログラムの箇所が一目でわかります。

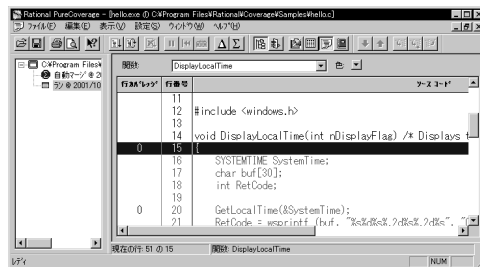
# PurifyPlus のその他の機能について

PurifyPlus の各ツールの詳細については、本書の該当箇所を参照してください。

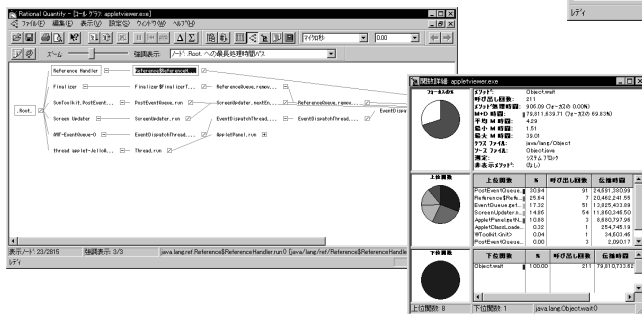
C/C++、Java、管理コードで検出が困難な障害を特定する方法については、7 ページの「Rational Purify ファースト ステップ」を参照してください。



未テスト コードを出荷しないようにする方法については、57 ページの「Rational PureCoverage ファースト ステップ」を参照してください。



パフォーマンスのボトルネックを強調表示する方法については、73 ページの「Rational Quantify ファースト ステップ」を参照してください。



Purify、Quantify、PureCoverage の各オンライン ヘルプでは、製品の使用方法や収集されたデータの解釈方法についての詳細な説明を参照できます。

Rational とほかの Rational 製品の詳細については、<http://www.rational.co.jp> を参照してください。

## Rational 技術サポートの連絡先

Rational 技術サポートへは、[support@japan.rational.com](mailto:support@japan.rational.com) (日本語対応化) または [support@apac.rational.com](mailto:support@apac.rational.com) (英語のみ対応) まで電子メールでお問い合わせいただけます。

また、インターネットまたは電話を利用してご連絡いただくことも可能です。連絡方法や、Purify、Quantify、PureCoverage に関してよく寄せられる質問に対する回答については、<http://www.rational.co.jp/supports/> を参照してください。



# Rational Purify

## ファースト ステップ

Purifyを使用すると、Visual C/C++ ネイティブ コード、Java、.NET 管理コードのどれで作業しても、より短時間で高品質のコードを開発できます。

### Visual C/C++ の開発者とテスターの皆さんへ

---

#### Purify と Visual C/C++: 主な機能

メモリ リークなどのランタイム エラーは発生箇所を突き止めるのが困難ですが、必ず修正しておく必要があります。メモリの不正使用の症状は予測がつかないだけでなく、通常エラーの原因として見えにくいものです。この種のエラーは、発生後かなり時間が経過しないと症状が現れません。たまたま行った作業によって誘発されるまでは表面化しないことすらあります。

Purifyを使用すると、こうした問題を簡単に解決できます。Purify の主な機能は次のとおりです。

- Visual C/C++ プログラムのランタイム エラーを迅速かつ総括的に検出します。
- ソース コードの有無にかかわらずエラーを検出します。
- 未テスト コードを表示するコード カバレッジ データを提供します。

Purify は自動的に Microsoft Visual Studio に統合されます。特別なビルドを作成する必要もなく、今までの作業過程を変更せず使用できます。

#### エラーを発生前に検出する

Purify では、メモリに関する広範なエラーが発生前に検出されるので、エラーをすばやく修正できます。Purify でチェックできるメモリに関する主なエラーの例は、次のとおりです。

- 配列の境界を越えたメモリに関するエラー
- 不定なポインタによるアクセス

- 初期化されていないメモリの読み込み
- メモリ割り当てに関するエラー
- メモリ リーク

**詳細情報:** Visual C/C++ 内で検出されるエラーのリストを参照するには、Purify の [ヘルプ] メニューの [Purify メッセージ] をクリックしてください。

## プログラム内のすべてのコンポーネントのチェック

今日のソフトウェア業界では、高品質のコンポーネントをいかに開発するかが、厳しい競争を勝ち抜くカギとなっています。高品質のアプリケーションをスケジュールどおりに出荷するためには、開発中のコードにエラーがないことはもちろん、そのアプリケーションで使用されるコンポーネントのどの箇所に問題があるのかを、ソース コードがない場合でも的確に検出することが大切です。コンポーネント内で発生するエラーは、そのコンポーネントに予期しないデータがコードにより提供されたことが原因であると考えられます。このようなエラーを検出できるのは Purify だけです。Purify を使用してコンポーネントを修正し、アプリケーションの信頼性を向上させましょう。

Purify では、複雑な複数スレッドや複数プロセス アプリケーションを含むプログラム内のすべてのコンポーネントがチェックされます。チェックされるコンポーネントの例は、次のとおりです。

- Windows のダイナミック リンク ライブラリ (dll) と Microsoft Foundation Class Library などの dll
- Visual Basic アプリケーション、Microsoft Internet Explorer、Netscape Navigator、Microsoft Office の各アプリケーションに埋め込まれた Visual C または C++ コンポーネント
- Microsoft Excel と Microsoft Word のプラグイン
- OLE と ActiveX コントロールを使用する、COM 対応のアプリケーション

Purify では、Windows API 関数への呼び出しがチェックされます。GDI、各インターネット サービス、システム レジストリ、COM インターフェイス API 関数、OLE インターフェイス API 関数などがこれに含まれます。また、メモリ ハンドルやポインタなどのパラメータも確認できます。

## 隅々までエラーをチェックする

Purify では、開発中のプログラムの実行中に発生する重大なエラーを検出するだけでなく、プログラム コードのどの部分をテストしたかを示すカバレッジデータを収集できます。PureCoverage がインストールされている場合は、Purify のカバレッジデータ収集はランごとに自動的に行われます。カバレッジデータには、チェックしたコードの全体に対する割合が報告され、未テストの行と関数が表示されます。このデータを使用して、コード全体に散在するエラーを確実に検出できるので、見逃がした行や関数に潜むエラーを修正せずに出荷すること也不再となります。

**詳細情報:** Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「カバレッジデータ」を参照してください。

## Purify を開発早期から使用する

Purify を最大限に活用するために、コードが実行可能になった段階で Purify の使用を開始します。開発サイクルを通して定期的に継続して Purify を使用するとより効果的です。特に以下を行う際に役立ちます。

- **コードのチェックイン:** ほかのコード モジュールに影響を与えるコード内の障害をチェックしておきます。
- **夜間テスト:** Purify をテスト ルーチンに組み込んで、各モジュールの互換性を確認したり、コードの依存関係や障害を検出します。カバレッジデータを各ランで収集し、追加または変更されたすべてのコードがテストされていることを確かめます。
- **コード テスト:** サード パーティ提供のコードまたはほかのグループからのコードを、開発中のアプリケーションに組み込む前に確認します。

Purify を開発早期から継続して使用することで、クリーンで安定した製品をスケジュールどおりに出荷することができます。

## Purify と Visual C/C++: 基本手順

Purify を使用すると、次の簡単な手順で、より安定した C/C++ コードを開発できます。

- 1 Purify を使用して開発中のプログラムを実行し、次のデータを収集します。
  - エラー データ
  - コード カバレッジ データ
- 2 エラー データを分析してソース コードを修正します。

3 カバレッジ データを収集した場合は、分析して Purify していない部分のコードを検出します。

4 Purify を使用してプログラムを再実行します。

ここでは、Microsoft Visual Studio 6 と統合された Purify の用法について説明します。Purify は、これ以外の方法で使用することもできます。ほかの用法については、以下を参照してください。

- 23 ページの「Purify の単独使用」
- 24 ページの「コマンド ライン インターフェイスを使用して C/C++ コードをテストする」

## Purify を使用して C/C++ プログラムを実行する

Visual Studio でプロジェクトを開き、[Purify] ツールバーの Purify 統合機能ボタンをクリックして、Purify 統合機能をオンにします。

PureCoverage がインストールされている場合は、エラーとメモリ リークをチェックするだけでなく、カバレッジ データも収集するように Purify を設定します。

Purify の統合機能をオンにするには、このボタンをクリックします。



カバレッジ データを収集するには、このボタンをクリックします。

Visual Studio の [ビルド] メニューのコマンドを使用して、プログラムをビルドして実行します。Purify で最も詳細なエラー報告とカバレッジ データを取得するには、プログラムをデバッグ データと再配置データ付きでビルドします。

**詳細情報:** デバッグ データと再配置データ付きでプログラムをビルドする方法については、Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「デバッグ データ」を参照してください。

Purify では、まずプログラムとそのプログラムが呼び出す各ライブラリのコピーが作成されます。次に、オブジェクト コード挿入 (OCI) 技術により、そのコピーがインストールメントされます。インストールメンテーションでは、メモリの読み込み、書き込み、割り当てと解放を確認する命令が挿入されます。カバレッジ データを収集する場合は、各行と関数の実行状況をカウントする命令も挿入されます。

各モジュールのインストールメンテーションの進行状況は、次のように表示されます。



Purify では、デフォルト設定のインストールメンテーション レベルで各モジュールがインストールされます。プログラムの特定の部分にフォーカスするには、デフォルトの設定を無効にして、インストールメンテーション レベルをカスタマイズします。

**詳細情報:** インストールメンテーション レベルの説明とその用法については、21 ページの「インストールメンテーションのカスタマイズ」を参照してください。詳細については、Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「インストールメンテーション」を参照してください。

Purify では、インストールされたモジュールが **cache** フォルダに保存されます。この保存されたモジュールを使用することで、プログラムの再実行時に時間とリソースを節約することができます。モジュールは、前回のラン以降に変更が追加された場合にのみ再度インストールされます。

プログラムのテストを実行すると、ランタイム エラーとメモリ リークが Purify によって検出され、[データ ブラウザ] ウィンドウの [エラー表示] タブに表示されます。

Purify の [データ ブラウザ]  
ウィンドウの [エラー表示]  
タブ



**詳細情報:** Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「エラー表示」を参照してください。

**メモ:** Purify では、クライアント/サーバーとマルチプロセス アプリケーションへの適用も簡単です。いくつかのプロセスを一度デバッグして、実行しているすべてのアプリケーションで発生しているエラーの報告を同時にチェックできます。各プロセスについて Visual Studio をそれぞれ起動し、Purify の統合機能をオンにしてそのプロセスを実行します。Purify のユーザー インターフェイスを単独使用することもできます。23 ページの「Purify の単独使用」を参照してください。

## プログラム上の問題を全体的に把握する

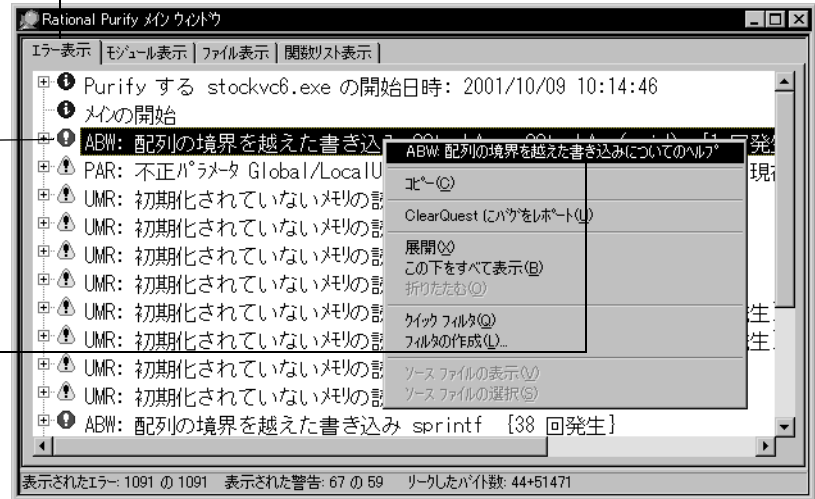
Purify では、プログラムの実行状況に関する情報メッセージと、ランタイムエラーとメモリ リークを報告するメッセージが表示されます。

メッセージの深度がここに表示されるアイコンの色でわかります。

❗ 情報メッセージ    ⚠ 警告メッセージ    ❗ エラー メッセージ

メッセージの種類が、ABW など、3 文字または 4 文字のアルファベットで表示されます。

メッセージに関する説明を表示するには、メッセージをマウスの右ボタンでクリックし、[<メッセージ>についてのヘルプ]をクリックします。



プログラムを終了すると、メモリ リークが報告されます。メモリ リークのほかに、使用中のメモリと使用中のハンドルをプログラム終了時に報告するように設定することもできます。

**詳細情報:** Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「エラーとリークの設定」を参照してください。

## 同一エラーが繰り返し発生する場合

ループの中などで同じエラーが繰り返し検出される場合があります。Purify では、診断報告をより読みやすくするために、デフォルトの設定により、初めて検出された問題のメッセージだけが表示されます。そのプログラムで同じ問題が繰り返し発生している場合は、その問題が発生するたびに、メッセージの右側にある発生回数が更新されます。

この「初期化されていないメモリの読み込み (UMR)」は 14 回発生しています。



**詳細情報:** エラーが発生するたびに、別個のメッセージが表示されるようにデフォルトの設定を変更することもできます。Purify オンラインヘルプの[トピックの検索]ウィンドウの[キーワード]タブでキーワード「エラーとリークの設定」を参照してください。

## 最重要エラーに最初に注目する

大きなプログラムでは、多数のメッセージが表示され、探しているメッセージを見つけにくいことがあります。そのような場合には、フィルタを作成して、あまり重要ではないメッセージを非表示にすることができます。

メッセージは、その種類や発生した箇所に従ってフィルタすることも可能です。たとえば、すべての情報メッセージや特定のファイルで発生した問題のメッセージなどをすべて非表示にすることができます。

フィルタしていないエラー表示では、プログラムで発生したすべてのメッセージが表示されます。

フィルタしたエラー表示では、指定したエラーメッセージのみが表示されます。


あるメッセージを自動的に非表示にするには、メッセージをマウスの右ボタンでクリックし、ショートカットメニューの[クイック フィルタ]をクリックします。

ショートカットメニューの[フィルタの作成]をクリックすると、フィルタするメッセージの基準を設定できます。



エラー フィルタは、無効にしない限り、作成後のあらゆるプログラム ランに適用されます。フィルタされて非表示になったメッセージは、そのフィルタを無効にするとエラー表示に再表示されます。

## エラー データ フィルタの使用

Purify のフィルタは、非常に便利で使用方法も簡単です。ツールバーの [フィルタ マネージャ] ボタン  をクリックすると、フィルタやフィルタ グループを作成して特定のプログラムまたはモジュールに適用できます。また、グローバル フィルタを作成して、実行するすべてのプログラムとモジュールに適用することもできます。Purify のフィルタは .pft ファイルに保存されるので、開発チームのメンバーと共有することも可能です。

フィルタのオン/オフを切り替えるには、ここをクリックします。

チェック ボックスがオンになっているフィルタは、チェック ボックスをオフにするか、そのフィルタを削除するまで、選択したプログラムに適用されます。

フィルタ マネージャでは、実行する各プログラムに適用するフィルタ グループを作成できます。

フィルタまたはフィルタ グループを移動またはコピーするには、そのフィルタまたはフィルタ グループをドラッグ アンド ドロップします。



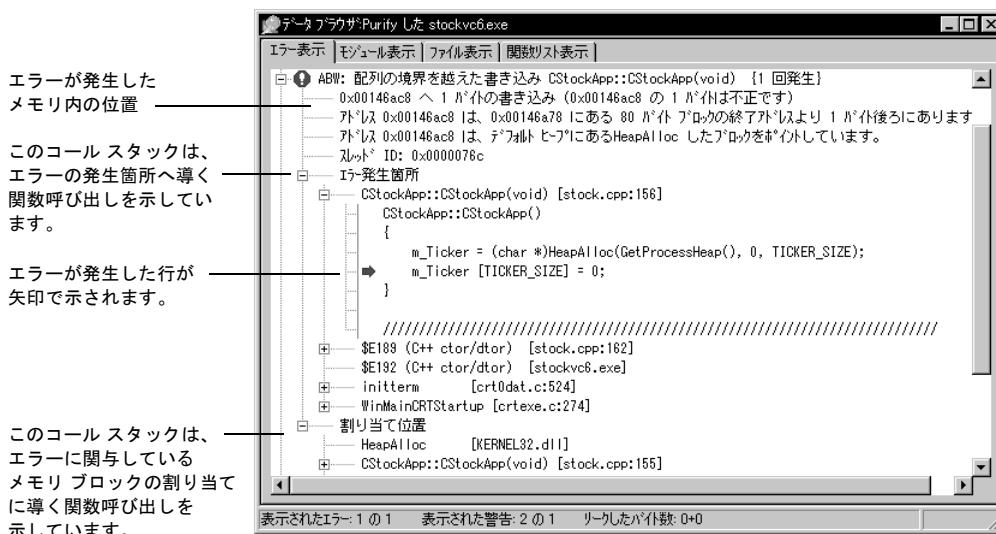
**詳細情報:** Purify では、エラー データだけでなくカバレッジ データもフィルタできます。Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「データのフィルタ」を参照してください。

フィルタのほかに、Purify には PowerCheck 機能が搭載されています。この機能では、特定のモジュールで発生しているエラーに注目できるだけでなく、インストールメンテーションの時間を短縮することもできます。PowerCheck 機能の詳細については、21 ページの「インストールメンテーションのカスタマイズ」を参照してください。

## Purify エラー データの分析

Purify のメッセージを展開すると、エラーの発生箇所が的確に指摘され、エラー発生の原因を分析するのに必要な診断情報が提供されます。

以下に、「配列の境界を越えた書き込み (ABW)」エラー メッセージの例を示します。



デバッグ データと再配置データの有無によって、エラー表示のコール スタックに表示されるデータの詳細度が異なります。プログラムをリリース モードでビルドした場合でも、可能な限り詳細なデータが表示されます。詳細については、Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「デバッグ データ、リリース バージョン」を参照してください。

Purify のメッセージのフォーマットをカスタマイズすることもできます。たとえば、表示するソース コードの行数を増やしたり、命令ポインタやオフセットを挿入してエラーの検出箇所をより簡単に示すようにしたりすることが可能です。

**詳細情報:** Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「基本設定、ソース コード」を参照してください。

## エラーの修正

Purify でのエラーの修正は簡単です。



**詳細情報:** Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「ソース コード、編集」を参照してください。

## Purify を使用してコード カバレッジをチェックする

コードに潜んでいるエラーを確実に検出するには、プログラムを実行するたびに Purify でコード カバレッジをモニターします。Purify のカバレッジ機能により、開発中のコードをすみずみまで実行し、テストしていることを確認できます。追加または変更したばかりの部分がテストされているかどうかを確認する場合には、特に便利な機能です。

各ビューに報告されているカバレッジデータは並べ替えることができ、一番大きな未テスト箇所がわかりやすく表示されます。

[モジュール表示] タブでは、関数がモジュールごとに一覧表示されます。

[ファイル表示] タブでは、関数がソース ファイルごとに一覧表示されます。

[関数リスト表示] タブでは、プログラムの各モジュールとファイルにあるすべての関数が一覧表示されます。

カバレッジデータを並べ替えるには、列見出しをクリックします。

関数を [コメント付きソースコード] ウィンドウに表示するには、その関数をダブルクリックします。

カバレッジ アイテム	呼び出し	未テスト 関数	テスト済み 関数	テスト済み 関数の%	未テスト行	テスト済...	テスト済み 行の%
CDialog::InitModalIndirect...	0	未テスト			5	0	0.00
CDialog::OnCancel(void)	0	未テスト			2	0	0.00
CDialog::OnCmdMsg(UINT...	0	未テスト			14	0	0.00
CDialog::OnCommandHelp...	0	未テスト			9	0	0.00
CDialog::OnCtlColor(CDC...	0	未テスト			2	0	0.00
CDialog::OnHelpHitTest(L...	0	未テスト			4	0	0.00
<b>CDialog::OnInitDialog(void)</b>	<b>0</b>	<b>未テスト</b>			<b>15</b>	<b>0</b>	<b>0.00</b>
CDialog::OnOK(void)	0	未テスト			4	0	0.00
CDialog::OnSetFont(CFont...	0	未テスト			1	0	0.00
CDialog::PostModal(void)	0	未テスト			10	0	0.00
CDialog::PreInitDialog(voi...	0	未テスト			1	0	0.00
CDialog::PreModal(void)	0	未テスト			8	0	0.00
CDialog::PreTranslateMes...	0	未テスト			12	0	0.00
CDialog::SetOncDialogInfr...	0	未テスト			3	0	0.00

Purify の [コメント付きソースコード] ウィンドウに行ごとのカバレッジ情報を表示することもできます。各行のステータス (テスト済み、未テスト、一部テスト済みなど) が色で識別できます。これにより、コードのどの部分を重点的にテストすればよいかが一目でわかります。

カバレッジ情報は、[コメント付きソースコード] ウィンドウにコードのコピーと共に表示されます。

各色の説明を表示するには、ここをクリックします。

この行は 1 度テストされました。

この行はテストされていません。

行番号	ソースコード	ステータス (色)
663	BOOL CDialog::OnInitDialog()	テスト済み (緑)
664	{	未テスト (白)
665	// execute dialog RT_DLGINIT resource	未テスト (白)
666	BOOL bDlgInit;	未テスト (白)
667	if (m_lpDialogInit != NULL)	未テスト (白)
668	bDlgInit = ExecutedlgInit(m_lpDialogInit);	未テスト (白)
669	else	未テスト (白)
670	bDlgInit = ExecutedlgInit(m_lpszTemplateName);	未テスト (白)
671		未テスト (白)
672	if (!bDlgInit)	未テスト (白)
673	{	未テスト (白)
674	TRACE0("Warning: ExecutedlgInit failed during dialc	未テスト (白)
675	EndDialog(-1);	未テスト (白)
676	return FALSE;	未テスト (白)
677	}	未テスト (白)

カバレッジデータに基づいて、コードのテスト状況を把握し、すべての重要な行と関数がテストされていることを確認します。テストを手動で行っている場合は、ほかのメニュー コマンドをテストしたり、新しい値を入力してテストします。自動テストの場合は、テスト スクリプトを編集または追加することで未テスト箇所を確実に実行するようにします。

**詳細情報:** Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「カバレッジ データ」を参照してください。

## プログラム ランの比較


エラー修正が一段落し、テストの最も必要な箇所を実行できるようになったら、プログラムをリビルドします。Purify を使用してプログラムを再実行します。

修正したプログラムを再実行した後は、各ランの診断情報を比較すると、エラーが修正されていることが容易に確認できます。Purify の [ナビゲータ] ウィンドウは、複数のプログラムやランの管理に便利です。[ナビゲータ] ウィンドウを表示するには、Purify の [ビュー] メニューを使用します。

[ナビゲータ] ウィンドウでは、ランがプログラムごとに一覧表示されます。

ランで検出された最も深刻度の高いメッセージが、ここに表示されるアイコンの色でわかります。



**詳細情報:** [ランの比較] ボタン  を使用して、異なるランのカバレッジ データを比較することができます。Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「ランの比較」を参照してください。

## Purify データの保存

Purify では、エラー データを保存して後で分析したり、ほかのチーム メンバーと共有したり、報告書などに添付したりできます。Purify のエラー データは次の形式で保存できます。

- **Purify データ ファイル (.pfy、.pcy):** ファイルの拡張子は、エラー データのみを保存するか、エラーとカバレッジ データを保存するかによって異なります。マージしたカバレッジ データを **PureCoverage** データ ファイル (.cfy) に保存することもできます。
- **ASCII テキスト ファイル (.txt):** スクリプトで処理したり、スプレッドシートまたはワープロ プログラムで開いたりすることができます。

**詳細情報:** Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「データの保存」を参照してください。

## Purify と Visual C/C++: 高度な機能

### インストゥルメンテーションのカスタマイズ

Purify では、各モジュールに対して、次の 2 つのエラー チェック用インストゥルメンテーション レベルがデフォルトの設定で指定されています。使用されるレベルは、該当のモジュールのサイズと、デバッグ データと再配置データの有無によって決定されます。

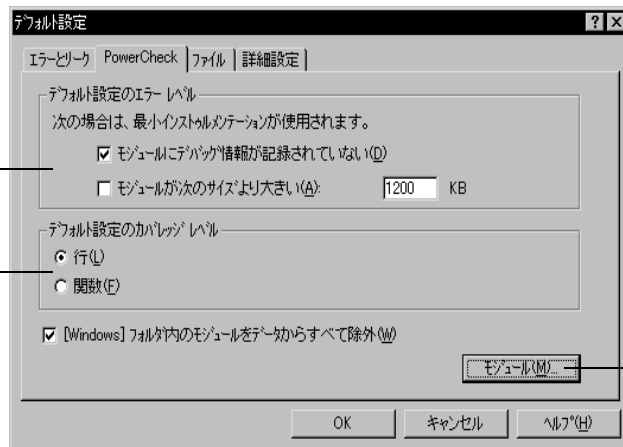
- **精密インストゥルメンテーション:** プログラム全体に潜む問題を指摘できる最高のランタイム エラー検出レベルです。
- **最小インストゥルメンテーション:** Purify のパフォーマンスが向上してインストゥルメンテーションの時間を短縮できるよう、基本レベルでエラーが検出されます。

カバレッジ モニターには、デフォルトの設定で次のインストゥルメンテーション レベルのいずれかが使用されます。

- **行レベル インストゥルメンテーション:** 行ごとのカバレッジ データを報告します。
- **関数レベル インストゥルメンテーション:** パフォーマンスは向上しますが、関数ごとのカバレッジ データのみが報告されます。

設定のダイアログボックスの [PowerCheck] タブでは、エラー検出のデフォルト設定のレベルを変更できます。

カバレッジ モニターのデフォルト設定のレベルも変更できます。



各モジュールのインストールメンテーション レベルを設定するには、ここをクリックします。

必要に応じて、各モジュールのインストールメンテーション レベルをそれぞれ指定することもできます。この場合、デフォルトの設定は無視されます。

リストから 1 つまたは複数のモジュールを選択します。


選択したモジュールに対するインストールメンテーション レベルを指定します。

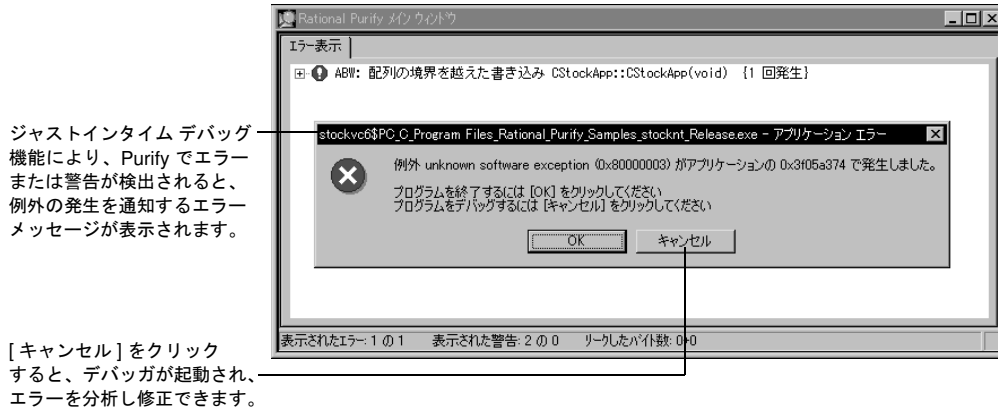


まず、プログラム内の最重要モジュールには精密インストールメンテーションを、その他のモジュールには最小インストールメンテーションを使用してみます。最重要モジュールのエラーを修正した後、最小インストールメンテーションを精密インストールメンテーションに変更すると、プログラム全体を能率的にチェックできます。

**詳細情報:** Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「インストールメンテーション レベル」と「PowerCheck の設定」を参照してください。

## ジャストインタイム デバッグ機能の使用法

Purify のジャストインタイム (JIT) デバッグ機能は、任意のデバッガを使用して厄介な問題を解決するときに便利です。[エラー発見時に中断]  をクリックすると、エラーの発生する直前にプログラムが中断され、デバッガが起動します。Purify したプログラムをデバッガを使用して実行することもできます。



プログラム内の最重要エラーをすばやくデバッグするには、[エラー発見時に中断] を Purify のエラー フィルタと共に使用します。フィルタによって、それほど重要でないメッセージをまず非表示にし、[エラー発見時に中断] をオンにします。フィルタされていないメッセージでのみプログラムが中断されます。非表示になっているエラーを修正するには、フィルタを無効にします。

**詳細情報:** Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「エラー発見時に中断」を参照してください。

## Purify の単独使用


Microsoft Visual Studio 6 のリソースの一部のみが必要な場合は、Purify を単独で 사용할こともできます。Visual Studio との統合機能を使用する場合と同様、Purify の使いやすい単独使用ユーザー インターフェイスでも、エラー検出とカバレッジデータ収集を効率よく行うことができます。

**メモ:** Purify の単独使用ユーザー インターフェイスは、Visual Studio との統合機能を使用しても利用可能です。Purify の [設定] メニューの [データブラウザの埋め込み] をオフにします。

単独使用の Purify を使用するには、[スタート] メニューから Purify を起動します。次に、表示される [ようこそ] ダイアログ ボックスで [実行] をクリックし、[プログラムの実行] ダイアログ ボックスを表示します。



コードがインストールされ、[データ ブラウザ] ウィンドウに診断結果が報告されます。

**詳細情報:** ツールバーのボタン、メニュー コマンド、ダイアログ ボックスの詳細については、 をクリックした後に該当アイテムをクリックします。

## コマンドライン インターフェイスを使用して C/C++ コードをテストする

Purify コマンドライン インターフェイスでは、既存の makefile、バッチ ファイル、Perl スクリプトと共に Purify を使用できます。たとえば、単にプログラムを実行するテスト スクリプトを、そのプログラムをインストールして実行するように変更するには、Exename.exe を実行する行を次のように変更します。

```
purify Exename.exe
```

テスト全体を通して、インストールしたバージョンの Exename.exe を実行するには、次の行をテスト スクリプトの冒頭に追加します。

```
purify /Replace=yes /Run=no Exename.exe
```

このコマンドを追加したテスト スクリプトでは、Exename.exe を .bak ファイルとして保存し、Exename.exe を実際に実行せずにインストールします。このテスト スクリプトはインストールしたバージョンの Exename.exe を実行し、詳細な診断情報を提供します。

プログラムをコマンドラインから実行し、エラー データと共にカバレッジ データを収集するには、次の /Coverage オプションを使用します。


```
purify /Coverage=yes Exename.exe
```

/SaveTextData オプションを使用すると、Purify をグラフィカル インターフェイスを使用しないで実行できます。このオプションによって、Purify の診断情報はテキスト形式で保存されます。保存したファイルのエラーと警告メッセージは、テスト結果の追加項目として使用できます。

**詳細情報:** Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「コマンドライン」を参照してください。

## Purify API 関数を使用してエラー チェックを強化する

Purify には、エラー チェック機能を強化し、エラーの追跡を容易にする API 関数が搭載されています。

Purify API 関数は、メモリ状態を設定または検査したり、メモリ リークやハンドル リークを検出するために使用します。たとえば、デフォルトの設定では、メモリ リークはプログラムの終了時まで報告されませんが、プログラムのキープointごとに API 関数 PurifyNewLeaks を呼び出して新規のメモリ リークを報告することもできます。プログラムの実行中に PurifyNewLeaks を呼び出すには、[NewLeaks] ボタン  をクリックするか、コードの重要箇所に PurifyNewLeaks を挿入します。その関数を最後に呼び出した以降に Purify で検出されたすべての新規メモリ リークが報告されます。定期的にプログラムをチェックすることにより、メモリ リークをより効率的に追跡できます。

プログラムを実行すると、Purify の [ビュー] メニューから Purify API 関数を呼び出すことが可能になります。Purify API 関数は、開発中のプログラムまたは Visual Studio 6 デバッガの [クイック ウォッチ] ダイアログ ボックスから呼び出すこともできます。

**詳細情報:** カバレッジ モニター用の関数を含むすべての Purify API 関数のリストを参照するには、Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「API 関数リスト」を参照してください。各関数の使用法の詳細については、Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「API 関数、使用法」を参照してください。

## Rational の統合の使用法

Rational の各ツールを既存の作業環境に統合すると、作業をより能率的かつ効果的に進めることができます。たとえば、Purify を、Rational の変更依頼管理ツールである Rational ClearQuest® (以下 ClearQuest) や、Rational の機能テスト ツールである Robot と Visual Test と共に使用できます。

## Purify を ClearQuest と共に使用する

ClearQuest が既にインストールされている場合は、Purify で検出されたエラーまたは警告や、カバレッジの問題をそのまま障害として報告できます。

エラーメッセージをマウスの右ボタンでクリックし、ショートカットメニューの [ClearQuest にバグをレポート] をクリックします。



ClearQuest の登録フォームの一部には、Purify によって自動的にデータが入力されています。また、エラーのカテゴリも自動的に指定されます。そのエラーの詳細な情報を追加する場合は、Purify データ ファイル (.pfy) を添付することもできます。

## Purify を Rational のテスト ツールと共に使用する

Robot が既にインストールされている場合は、Robot で再生オプションを設定して、Robot テスト スクリプトを実行するときに Purify のエラー データとリーク データを収集できます。Purify によりコードが実行されると、メモリ エラーが検出されます。Robot には、エラー データとリーク データ以外にコード カバレッジ情報を収集できる再生オプションもあります。

Visual Test が既にインストールされている場合は、Visual Studio 内で Visual Test により分析しているプログラムを Purify することができます。テスト ルーチンを使用して Visual Test スクリプトを実行する場合でも、プログラムのテスト中に自動的に Purify を実行できるように簡単に変更できます。

**詳細情報:** Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「ClearQuest」、「Robot」、「Visual Test」を参照してください。また、ClearQuest、Robot、Visual Test の各マニュアルも参照してください。



Purify を C/C++ コードで使用するための基礎知識はここからです。Purify の使用中に疑問が生じたときは、Purify のオンライン ヘルプを参照してください。オンライン ヘルプには、Purify の各機能について、より詳しい説明が記載されています。

---

## Java の開発者とテスターの皆さんへ

### Purify と Java: 主な機能

#### Java コードのメモリ リーク

Java コードでもメモリ リークは発生し、重大な結果をもたらす場合すらあります。

Java 仮想マシン (JVM) では、ガーベッジ コレクタによってプログラムで不要になったメモリ オブジェクトから自動的に除去されますが、ほかのプログラム上のコンテキストで発生するメモリ リークのほとんどが回避できます。ただし、Java アプリケーションは次第により多くのメモリを消費していきます。この原因を追跡することが非常に困難になることもあります。Purify では、この問題を簡単に検出して修正できます。

Java コードのメモリー リークでは、不要になったオブジェクトを参照している場合と、システム リソースが解放されていない場合の 2 つのカテゴリが考えられます。

#### 不要になったオブジェクトの参照

Java コードでは、不要になったメモリへの参照が残り、そのメモリがガーベッジ 収集時に収集されないということが頻繁に起こります。Java オブジェクトには通常、ほかのオブジェクトへの参照が含まれているので、1 つのオブジェクトがオブジェクト ツリー全体のオブジェクトをメモリに保持することもあります。たとえば、次のような場合には問題が発生することがあります。

- オブジェクトを配列に追加したまま放置した場合。
- 次に使用するまで、オブジェクトへの参照を保持した場合。たとえば、あるメニュー コマンドによってオブジェクトが作成され、該当のコマンドが次回呼び出されるまでそのオブジェクトへの参照が解放されない場合が考えられます。該当のコマンドが再度呼び出されない場合は、そのオブジェクトへの参照も解放されません。
- オブジェクトの状態が変更されたにもかかわらず、以前の状態を反映している参照がある場合。たとえば、テキスト ファイルのプロパティを配列に格納し、次にバイナリ ファイルのプロパティを格納すると、「文字数」などのフィールドが不要になったメモリを保持し続けることがあります。
- スレッドが長時間実行されているために参照が固定される場合。オブジェクト参照を NULL に設定しても、スレッドが終了するか待機状態になるまで、メモリはガーベッジ収集されません。

## システム リソースが解放されていない場合

Java メソッドが、ウィンドウやビットマップのリソースなど、Java インスタンス外にあるヒープ メモリを割り当てられる場合もあります。Java コードでは、Java Native Interface (JNI) の呼び出しを使用して C または C++ 言語のルーチン呼び出し、リソースをこのように割り当てることがよくあります。

## Purify を使用した処理

Purify では、ガーベッジ コレクタによって解放されないメモリのうち、相当量を占有しているメソッド、クラス、オブジェクトが報告され、Java のメモリリークを検出できます。

Purify で収集したデータを使用して、メモリの問題にフォーカスすることができます。メモリの問題が発生している箇所を突き止めたら、不要なオブジェクトへの参照を削除するか、コードの主要領域でガーベッジ収集を強制実行することが可能です。システム リソースを解放する方法については、Java ツールキットを参照してください。たとえば、Sun Microsystems Abstract Windowing Toolkit (AWT) の `dispose()` メソッドを使用すると、Frame、Dialog、Graphics の各クラスによって使用されるシステム リソースが解放されます。

メモリ プロファイリング データは、プログラム実行中いつでも収集できます。新しい機能をテストしてからコードをチェックインする場合は、Purify のグラフィカル ユーザー インターフェイスからコードを実行します。詳細については、29 ページの「Purify と Java: 基本手順」を参照してください。テスト ルーチンから自動的にデータを収集するには、テスト スクリプト内にある Purify のコマンドライン インターフェイスを使用して Purify API 関数呼び出しをコードに挿入し、データ収集を制御します。詳細については、40 ページの「Java テスト環境への Purify の統合」を参照してください。

**詳細情報:** Purify で過度なメモリ量の浪費を検出するだけでなく、アプリケーションのパフォーマンスとテスト カバレッジを向上させることも可能です。これには、PurifyPlus に含まれている PureCoverage と Quantify を使用します。PureCoverage を使用すると、コードで未テストの箇所が報告されます。Quantify を使用すると、コードのパフォーマンスを低下させるボトルネックを検出できます。詳細については、57 ページの「Rational PureCoverage ファースト ステップ」と 73 ページの「Rational Quantify ファースト ステップ」を参照してください。

## Purify と Java: 基本手順

不要なオブジェクトへの参照が残っているためオブジェクトをガーベッジ収集できない場合、Java アプリケーションの実行中に消費するメモリは次第に大量になっていきます。Purify では、Java プログラムのメモリ使用量を測定し、これらの「メモリ リーク」の原因になっているオブジェクトを的確に検出します。また、強制ガーベッジ収集によってコードのパフォーマンスが向上する箇所も指摘できます。

Purify を使用して Java のメモリ使用状況をプロファイルするには

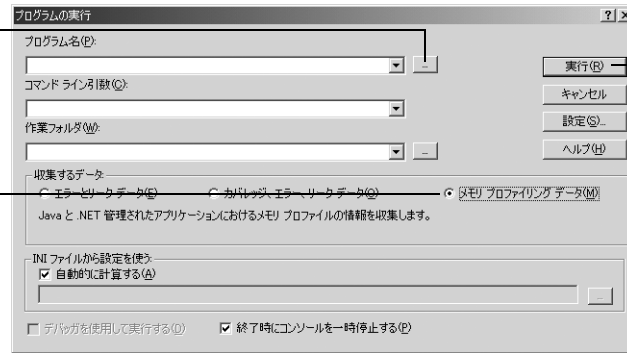
- 1 Purify を使用して Java プログラムを実行します。
- 2 初期化完了後、スナップショットを撮ります。
- 3 リークしていると思われるコードを実行し、再度スナップショットを撮ります。
- 4 2 つのスナップショットを比較して、メモリの問題の原因になっていると思われるメソッドを特定します。
- 5 これらのメソッドによって割り当てられ、リークしたオブジェクトを検出し、オブジェクトがガーベッジ収集されないようにしている参照を特定します。

## Purify を使用して Java プログラムを実行する

Java プログラムを Purify するには、Purify を起動して [ようこそ] ダイアログボックスの [実行] をクリックします。[プログラムの実行] ダイアログボックスが表示されます。

[...] ボタンをクリックして、プロファイルする Java プログラム、アプレット、クラス、JAR ファイルを選択します。

メモリ プロファイリングデータを収集するために、[メモリ プロファイリングデータ] を選択します。




最後に [実行] をクリックします。

**詳細情報:** Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「JVM の指定」と「プログラムの実行」を参照してください。

プログラムのランが進行するにしたがって、JVM からのメモリ使用状況に関するメッセージがインターセプトされ表形式で報告されます。これらのメッセージに基づいて、プログラムによって各メソッドとオブジェクトに割り当てられたメモリ量が逐次追跡されます。

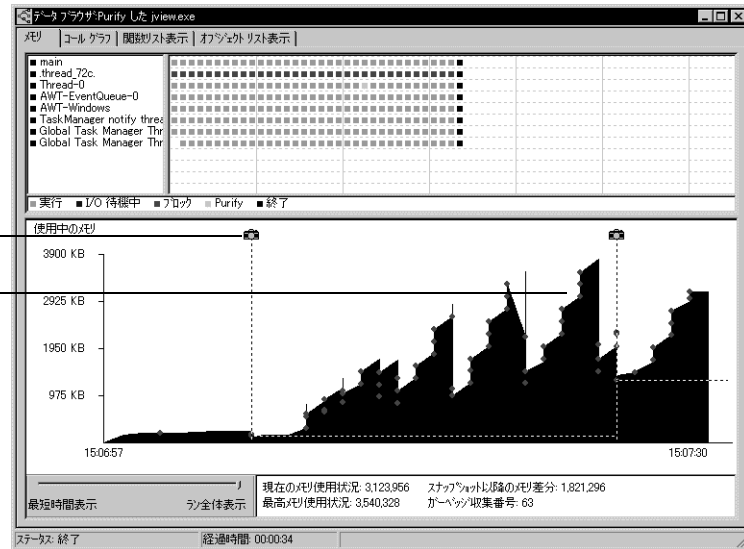
## メモリ使用状況のスナップショットを撮る


Java プログラムのメモリ リークにフォーカスして分析するには、プログラムの初期化プロシージャ完了後、 をクリックして現在のメモリ使用状況のスナップショットを撮ります。このスナップショットは、プログラム実行時のメモリ使用状況を分析するためのベースラインです。

メモリをリークしていると思われるプログラム ルーチンを実行します。プログラムのランが進行するにしたがって、Purify の [データ ブラウザ] ウィンドウの [メモリ] タブに、プログラムのメモリ使用状況を示すグラフが表示されます。

プログラムの初期化完了後、最初のスナップショットを撮ります。

メモリ使用量が増加していることを確認し、再度スナップショットを撮ります。




グラフのメモリ使用状況の変化に注意します。メモリ使用量が著しく増加した場合、特に  をクリックして強制的にガーベッジ収集してもメモリ使用量が減少しないときなどは、問題が潜んでいる可能性があります。

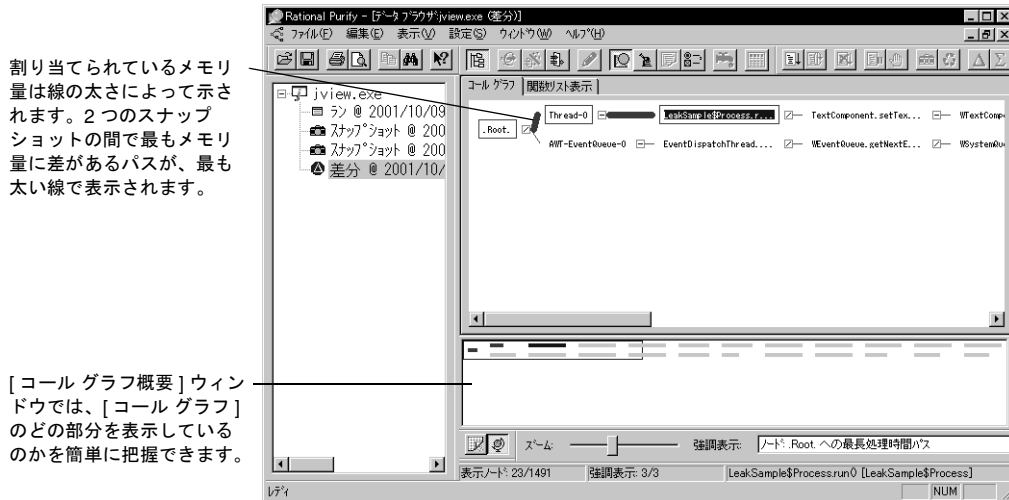
プログラムの実行前後のメモリ使用状況を記録するため再度スナップショットを撮り、プログラムを終了します。

**詳細情報:** Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「スナップショットの作成」と「ガーベッジ収集」を参照してください。

## スナップショットを比較して重要なコードにフォーカスする

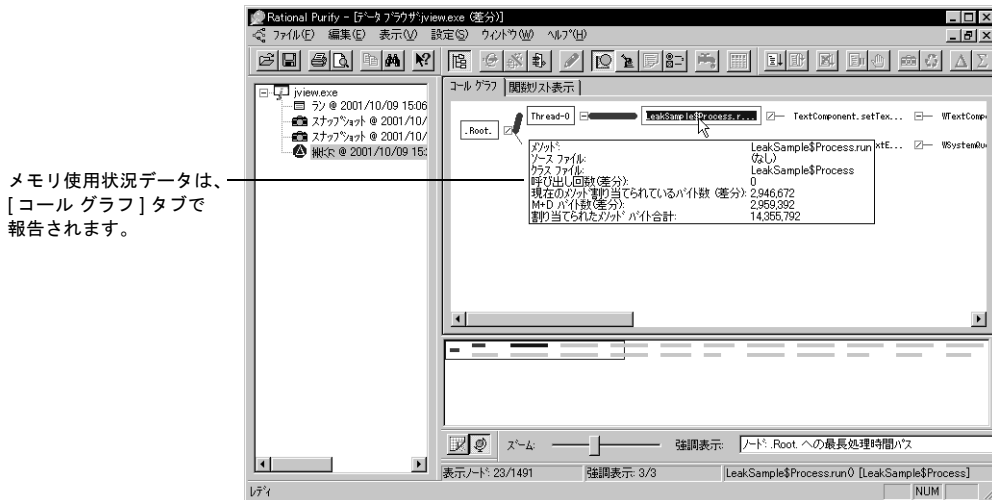
[ナビゲータ] ウィンドウで2番目のスナップショット エントリを選択し、 をクリックして、最初のスナップショットと比較します。

[コール グラフ] タブが表示され、最初のスナップショットと 2 番目のスナップショット間に最も多くのメモリを割り当てたメソッドが報告されます。



[コール グラフ] タブには、メソッド間の呼び出し関係も表示されます。このデータにより、不要なオブジェクトへの参照が残り、ガーベッジ収集が不可能になっているメソッドが検出できます。

分析するメソッドまたはパスの上にカーソルを移動します。ツールチップが表示され、そのメソッドのメモリに関連したデータが表示されます。



このデータにより、メモリを浪費しているメソッドとその下位メソッドにフォーカスして分析できます。

Purify でソース コードを表示するには、ソース コードが利用可能なメソッドをマウスの右ボタンでクリックし、ショートカットメニューの[ソース ファイル]をクリックします。

**詳細情報:** Purify オンライン ヘルプの[トピックの検索]ウィンドウの[キーワード]タブでキーワード「スナップショットの差分」、「コール グラフ」、「ソース コード」、「表示」を参照してください。

## [関数リスト表示] タブを使用してリークを診断する

[データ ブラウザ] ウィンドウの[関数リスト表示]タブには、同じデータがテキストで非階層表示されます。[関数リスト表示]タブでは、プログラム全体のデータを並べ替えて、メモリの消費量が最も多いメソッドを検出できます。

メモリ プロファイリングデータを並べ替えるには、列見出しをクリックします。

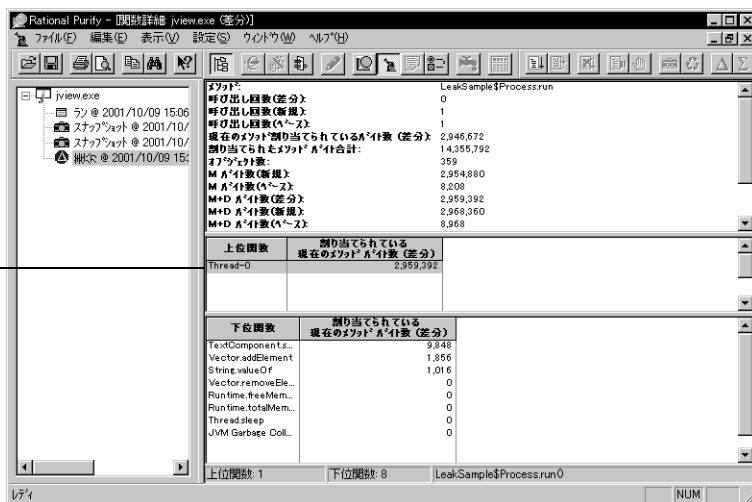
メソッド	呼び出し回数 (高減)	呼び出し回数 (低減)	現在のメソッドに割り当てられているメソッド数 (差分)	クラス ファイル
LeakSample\$Process.run0	1	1	2,946,672	com.ms.awt.WComponentPeer
com.ms.awt.WComponentPeer	502	115	3,328	com.ms.awt.WComponentPeer
Vector.ensureCapacity	5	1	1,856	java.util.Vector
Timer.schedule	629	16	1,440	com.ms.util.Timer
TaskManager.schedule	631	20	1,296	com.ms.util.TaskManager
StringBuffer.<init>	332	16	696	java.lang.StringBuffer
GraphicsX.intersect	1,828	302	624	com.ms.awt.GraphicsX
FormattedText...	320	8	576	com.ms.awt.FormattedText
UIStateContainer...	322	4	240	com.ms.ui.UIStateContainer
StringBuffer.toString	378	12	240	java.lang.StringBuffer
UIStateComponent...	332	16	192	com.ms.ui.UIStateComponent
WJPeer.convert...	336	15	96	com.ms.awt.WJPeer
FormattedText...	320	12	96	com.ms.awt.FormattedText
UIStateComponent...	476	42	80	com.ms.ui.UIStateComponent
UIStateContainer...	634	24	80	com.ms.ui.UIStateContainer
Long.toString	320	2	80	java.lang.Long
UIContainer.initialize	394	32	72	com.ms.ui.UIContainer
PaintCache.initialize	176	94	72	com.ms.ui.windowmanager.PaintCache
UIDrawText.setText	372	68	64	com.ms.ui.UIDrawText
util.<init>	6	0	60	com.ms.util
Timer.addTimerList	7	6	56	com.ms.util.Timer
Vector.<init>	13	12	48	java.util.Vector
WToolkit.createNa...	502	115	48	com.ms.awt.WToolkit

**詳細情報:** Purify オンライン ヘルプの[トピックの検索]ウィンドウの[キーワード]タブでキーワード「[関数リスト表示]タブ」を参照してください。

## [関数詳細] ウィンドウを使用してメソッドにフォーカスする

[コール グラフ] または [関数リスト表示] で任意のメソッドをダブルクリックすると、[関数詳細] ウィンドウが表示されます。このウィンドウには、選択したメソッド、その上位メソッド、その下位メソッドがメモリをどのように割り当てたかが報告されます。

[上位関数] 列または  
[下位関数] 列にある  
メソッドのいずれかを  
ダブルクリックすると、  
そのメソッドのメモリ  
データが表示されます。



**詳細情報:** Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「[関数詳細] ウィンドウ」を参照してください。

あるメソッドが割り当てたメモリ量が予想以上に多い場合はソース コードを調べます。別のメソッドや下位メソッドの参照しているオブジェクトがメモリのガーベッジ収集を妨げていることも考えられます。たとえば、下位メソッドに文字列配列の一部として静的変数が含まれていると、その配列全体のメモリがスコープ内にとどまり、プログラムの速度を低下させたりプログラムが応答しなくなる原因になります。

メモリの問題の原因になっていると思われるメソッドが見つかった場合は、メソッドのオブジェクトを調べます。Purify では、メソッドだけでなく、プログラム内のすべてのオブジェクトとそのメモリ使用状況に関するさまざまな情報が提供されます。

## 不要なオブジェクトの検索

プログラムで不要になったオブジェクトによりメモリがガーベッジ収集されず、その結果プログラムの速度が次第に低下していくことはよくあることです。Purify では、いろいろな形式でオブジェクトの総合メモリ データが報告されるので、この種の問題を簡単に追跡することができます。

**メモ:** オブジェクト データを分析するには、スナップショットまたは集成ランデータを使用します。▲ をクリックすると作成される比較データには、オブジェクト データは含まれていません。

## 疑わしいメソッドからそのオブジェクトまでのデータを取得する

[関数詳細] ウィンドウには、メソッドに関する情報に加え、メソッドによって割り当てられたオブジェクトも一覧表示されます。列見出しをクリックすると、リスト内のオブジェクトを並べ替えることができます。

メソッドが現在割り当てられているオブジェクト。オブジェクトをダブルクリックすると、[オブジェクト詳細] ウィンドウにそのオブジェクトの総合メモリ データが表示されます。

スナップショット用の [関数詳細] ウィンドウには、メモリ割り当てを示す円グラフも表示されます。

オブジェクトの名前	クラスの名称	サイズ	O+R サイズ	割合
TextField 211...	TextField	188	188	
Button 212...	Button	164	1,868	
char [6] 212...	char []	12	12	
String 2121...	String	12	24	

上位関数	呼び出し回数	割り当てられて現在のメソッド
LeakSample.main	1	

下位関数	呼び出し回数	割り当てられて現在のメソッド
TextField.<init>	2	
Window.show	1	
Frame.<init>	1	
Button.addActionListener	2	
ClassLoader.loadClass	12	
TextField.<init>	1	

上位関数: 1    下位関数: 26    LeakSample.<init> (java.lang.String)

レディ    NUM

# オブジェクト詳細の分析

[関数詳細] ウィンドウでオブジェクトのいずれかをダブルクリックすると、[オブジェクト詳細] ウィンドウが表示されます。このウィンドウには、選択したオブジェクトのメモリ関連情報がすべて表示されるので、大量のメモリを保持しているオブジェクトを特定してその存続期間を決定できます。

オブジェクト参照グラフでは、現在のオブジェクトを参照しているオブジェクトと、現在のオブジェクトによって参照されているオブジェクトがそれぞれ表示されます。

あるオブジェクトの詳細なメモリ情報を表示するには、そのオブジェクトにカーソルを移動して停止します。

オブジェクト参照グラフで強調表示するオブジェクトの種類を選択します。

サイズや作成時刻など、オブジェクト参照グラフで現在選択されているオブジェクトの詳細データ

オブジェクト参照グラフ

.Root

String 2120F988 char [2] 2120F980

String 211CDCB8 char [2] 211CDCB0

Class 2120F8F0

Object ID: String 211CDCB8 8] 2120F998

クラス名前: String

スレッド名前: Thread-1 8] 2120F9D0

サイズ: 12

O+R Size: 16

参照: 1

参照先: 1

ガーベッジ収集不可回数: 20

作成時間: 14:48:08

オブジェクト名前:	Class 2120F8F0	Name	Value
クラス名前:	Class	int EXECUTE	1
スレッド名前:	Thread-1	int WRITE	2
サイズ:	140	int READ	4
O+R Size:	268	int DELETE	8
ガーベッジ収集不可回数:	20	int ALL	15
作成時間:	14:48:10	int NONE	0
行番号:	0	java/lang/String REC...	2120F988
参照:	4	java/lang/String WLD	211CDCB8
参照先:	0	java/lang/String SEP...	2120F9C0
ルートの型:	System Class	java/lang/String SEP...	2120F9F8

参照元: 4

参照先: 0

java/lang/Class 2120F8F0

## 割り当てられたすべてのオブジェクトを確認する

プログラムのトップ レベルのオブジェクトを表示するには、メモリの問題があると思われるスナップショットの [データ ブラウザ] ウィンドウを開き、[オブジェクト リスト表示] タブをクリックします。

リストのアイテムを並べ替えるには、列見出しをクリックします。

プログラムで現在割り当てられているすべてのトップ レベル オブジェクトのメモリ データ

ステータス バーには、選択された行番号とオブジェクトの合計数が表示されます。

オブジェクト名	クラス名	メソッド名	サイズ	O+R Size	ガーベッジ収集不可回数	作成時間	
byte [8196] 212B6DD0	byte []	LeakSample\$Proc...	8,196	8,196	4	14:48:35	
LeakSample 21270B88	LeakSample	LeakSample.main	300	1,868	18	14:48:11	
TextField 2124160	TextField	LeakSample.<init>	188	188	17	14:48:12	
TextField 21236910	TextField	LeakSample.<init>	188	188	18	14:48:11	
Class 21220010	Class	Properties.load	140	140	18	14:48:11	
Class 212200A0	Class	Properties.load	140	140	18	14:48:11	
Class 212400D0	Class	Event.<clinit>	140	140	17	14:48:13	
Class 21220130	Class	Properties.load	140	140	18	14:48:11	
Class 212201C0	Class	Class.forName0	140	140	18	14:48:11	
Class 212204B8	Class	CustomComponent...	140	140	17	14:48:14	
Class 21220590	Class	CustomComponent...	140	312	17	14:48:14	
Class 211E05F0	Class	Thread-1	140	140	18	14:48:09	
Class 211E06D0	Class	Thread-1	140	140	18	14:48:09	
Class 212206D8	Class	RasterOutputMana...	140	144	17	14:48:14	
Class 211E0760	Class	Thread-1	140	21,888	18	14:48:09	
Class 21240768	Class	Component.dispatc...	140	140	17	14:48:13	
Class 21220820	Class	GeneralRenderers...	140	280	17	14:48:14	
Class 21240898	Class	InputContext.getIn...	140	2,796	17	14:48:13	
Class 21240928	Class	InputContext.getIn...	140	140	17	14:48:13	
Class 21220988	Class	GeneralRenderers...	140	328	17	14:48:14	
Class 212409B8	Class	InputContext.getIn...	140	140	17	14:48:13	

オブジェクト: 4/568    java/awt/TextField 212369 LeakSample.<init> (java.lang.String)  
メモリ: NUM

スナップショットを撮ったときに割り当てられていたすべてのトップ レベルオブジェクトが一覧表示されます。オブジェクトのリストには、オブジェクトのサイズ、オブジェクトの作成時刻、ガーベッジ収集ができなかった回数などいろいろな情報が表示されます。リストを並べ替えると、最も多くのメモリを保持しているオブジェクトと最も古いオブジェクトを簡単に検索できます。

リストのオブジェクト名をダブルクリックすると、そのオブジェクトの [オブジェクト詳細] ウィンドウが表示されます。

不要になったと思われるオブジェクトを検索する場合は、ソース コードを参照します。オブジェクトが実際に不要であることがわかった時点で、そのオブジェクトへのすべての参照を解放してガーベッジ収集できるようにコードを変更します。

**詳細情報:** Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「[関数詳細] ウィンドウ」、「[オブジェクト詳細] ウィンドウ」、「[オブジェクト リスト表示] タブ」を参照してください。

## Purify メモリ プロファイリング データの保存

Purify のデータを保存して後で分析したり、ほかのチーム メンバーと共有したり、報告書などに添付したりすることができます。Purify の Java データは次の形式で保存できます。

- Purify メモリ プロファイリング ファイル (.pmy): ラン、スナップショット、ほかのデータの場合と同じように、Purify で表示できるファイルです。
- ASCII テキスト ファイル (.txt): スクリプトで処理したり、スプレッドシートまたはワープロ プログラムで開いたりすることができます。

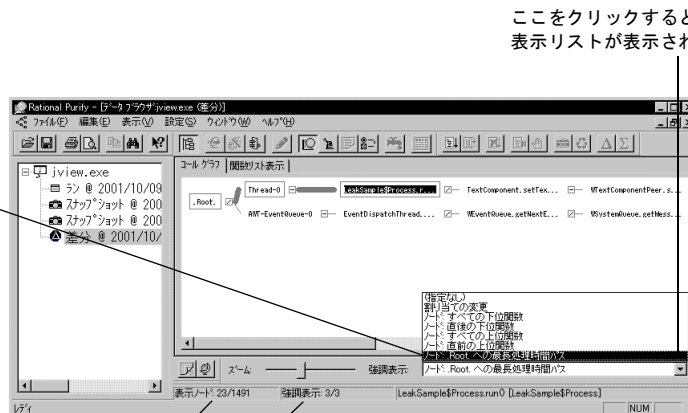
**詳細情報:** Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「データの保存」を参照してください。

## Purify と Java: 高度な機能

### 同一の主要属性を持つメソッドの強調表示

特定のメモリ関連のデータまたは呼び出し関係を表示する場合は、該当するメソッドを [コール グラフ] で強調表示します。

たとえば、[ノード: Root への最長処理時間パス] をクリックすると、最も多くのメモリが割り当てられているパスで、選択したメソッドと .Root との間にあるすべてのメソッドが強調表示されます。



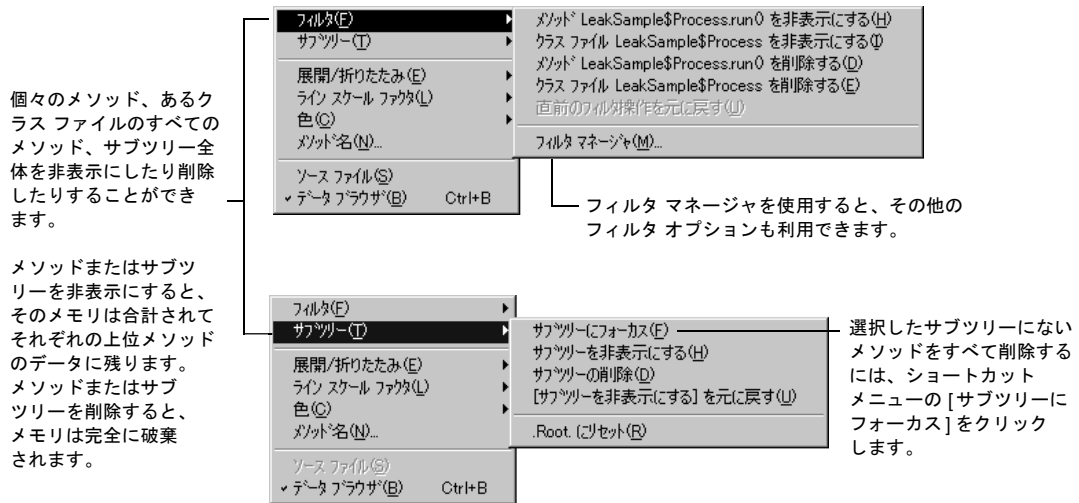
現在のデータにある 1491 の関数のうち 23 の関数が [コール グラフ] に表示されています。

.Root への最長処理時間パスにある 3 つの関数がすべて [コール グラフ] に表示されています。

**詳細情報:** Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「強調表示」を参照してください。

## データのフォーカス

Purify のフィルタ関連のコマンドを使用すると、選択したメソッド、またはあるクラス ファイル内のすべてのメソッドを **Purify** データで非表示にすることができます。または、サブツリー関連のコマンドを使用して、特定のメソッドとその下位メソッドにフォーカスしたり、**Purify** データから非表示にすることもできます。フィルタとサブツリー関連のコマンドは、[ コール グラフ ]、[ 関数リスト表示 ]、関数詳細のいずれかでメソッド名をマウスの右ボタンでクリックすると表示されるショートカット メニューにあります。



Purify には、すべてのフィルタとサブツリー関連のコマンドの操作を元に戻す機能があり、データをコマンド適用前の状態に戻すことが簡単にできます。

[ コール グラフ ] には、サブツリーでの作業に便利なデータの展開と折りたたみ関連のコマンドもあります。しかし、フィルタとサブツリー関連のコマンドとは異なり、これらのコマンドは [ コール グラフ ] に表示されているデータ表示にのみ影響します。データ自体は変更されません。

メニュー コマンドのほかに、フィルタ マネージャを使用して必要なデータを選択する方法もあります。

クラス ファイルまたはメソッドごとにデータをフィルタできます。

フィルタのオン / オフを切り替えるには、ここをクリックします。



**詳細情報:** Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「データのフィルタ」と「サブツリー」を参照してください。

## Java テスト環境への Purify の統合

Purify コマンドライン インターフェイスを使用すると、自動テスト環境でメモリ プロファイリング データを収集することができます。既存の **makefile**、バッチ ファイル、Perl スクリプトを変更し、Purify でプログラムを実行します。たとえば、Java クラス ファイルを実行するテスト スクリプトで、Sun Microsystem の Java ビューアを使用する場合は、クラス ファイルを実行する行を次のように変更します。

```
Purify /SaveData Java Java.exe Classname.class
```

このコマンドでは、クラス ファイルを実行してメモリ プロファイリング データを収集し、そのデータを **.pmf** ファイルに保存します。このファイルは、Purify インターフェイスで開いて分析したり、ほかのチーム メンバーと共有することができます。

データを **.txt** ファイルに保存する場合は、**/SaveData** オプションではなく **/SaveTextData** オプションを使用します。このデータを処理するスクリプトを作成して、プログラムのメモリ使用状況に関するレポートを生成することができます。たとえば、現在の夜間テストのデータと、以前の夜間テストのデータを比較するときに、メモリ関連のリグレーションを発生と同時に検出します。

自動データ収集を制御して、比較可能なデータを各テストから確実に生成するには、Purify API を使用します。次の「Purify API を使用して Java メモリ プロファイリングを制御する」を参照してください。

**詳細情報:** Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「コマンドライン」を参照してください。

## Purify API を使用して Java メモリ プロファイリングを制御する

Purify には、メモリ プロファイリング機能を強化する API 関数が搭載されています。

API は、自動テストを行う場合に特に便利です。API 関数を使用して、プロファイルされている部分のコードを判別し、プログラムの初期化動作を除外して特定のモジュールまたはルーチンにフォーカスすることができます。初期化完了後にデータを消去して、プログラムの実行と同時にデータ収集を継続し、プログラムの終了直前にデータを保存することもできます。これは、Purify ユーザー インターフェイスで 2 つのスナップショットを比較する場合と同じことです。

**詳細情報:** カバレッジ モニター用の関数を含むすべての Purify API 関数のリストを参照するには、Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「API 関数リスト」を参照してください。各関数の使用法の詳細については、Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「API 関数、使用法」を参照してください。



Purify を Java コードで使用するための基礎知識はここまでです。Purify の使用中に疑問が生じたときは、Purify のオンライン ヘルプを参照してください。オンライン ヘルプには、Purify の各機能について、より詳しい説明が記載されています。

## .NET 管理コードの開発者とテスターの皆さんへ

---

### Purify と .NET 管理コード: 主な機能

Purify では、.NET 管理コード (アセンブリ、.exe ファイル、.dll ファイル、OLE/ActiveX コントロール、COM オブジェクト) 内のメモリ リークが検出され、報告されます。方法は、Java の場合と同様です。したがって、以下の説明は Purify と Java を使用した場合と類似した内容になっています。

#### 管理コード内のメモリ リーク

管理コードからメモリがリークして、プログラムが正常に動作しないことがあります。

.Net 管理コードでは、ガーベッジ コレクタによってプログラムで不要になったメモリ オブジェクトから自動的に除去されますが、ほかのプログラム上のコンテキストで発生するメモリ リークのほとんどが回避できます。ただし、管理コード アプリケーションは次第により多くのメモリを消費していきます。この原因を追跡することが非常に困難になることもあります。Purify では、この問題を簡単に検出して修正できます。

管理コードのメモリー リークでは、不要になったオブジェクトを参照している場合と、システム リソースが解放されていない場合の 2 つのカテゴリが考えられます。

#### 不要になったオブジェクトの参照

管理コードでは、不要になったメモリへの参照が残り、そのメモリがガーベッジ収集時に収集されないということが頻繁に起こります。管理コード オブジェクトには通常、ほかのオブジェクトへの参照が含まれているので、1 つのオブジェクトがオブジェクト ツリー全体のオブジェクトをメモリに保持することもあります。たとえば、次のような場合には問題が発生することがあります。

- オブジェクトを配列に追加したまま放置した場合。
- 次に使用するまで、オブジェクトへの参照を保持した場合。たとえば、あるメニュー コマンドによってオブジェクトが作成され、該当のコマンドが次回呼び出されるまでそのオブジェクトへの参照が解放されない場合が考えられます。該当のコマンドが再度呼び出されない場合は、そのオブジェクトへの参照も解放されません。
- オブジェクトの状態が変更されたにもかかわらず、以前の状態を反映している参照がある場合。たとえば、テキスト ファイルのプロパティを配列に格納し、次にバイナリ ファイルのプロパティを格納すると、「文字数」などのフィールドが不要になったメモリを保持し続けることがあります。

- スレッドが長時間実行されているために参照が固定される場合。オブジェクト参照を `NULL` に設定しても、スレッドが終了するか待機状態になるまで、メモリはガーベッジ収集されません。

## システム リソースが解放されていない場合

管理コードメソッドが、ウィンドウやビットマップのリソースなど、管理データインスタンス外にあるヒープメモリを割り当てられる場合もあります。管理コードでは、C または C++ 言語のルーチン呼び出し、リソースをこのように割り当てることがあります。

## Purify を使用した処理

Purify では、ガーベッジコレクタによって解放されないメモリのうち、相当量を占有しているメソッド、クラス、オブジェクトが報告され、管理コードのメモリリークを検出できます。

Purify で収集したデータを使用して、メモリの問題にフォーカスすることができます。メモリの問題が発生している箇所を突き止めたら、不要なオブジェクトへの参照を削除するか、コードの主要領域でガーベッジ収集を強制実行することが可能です。

**詳細情報:** Purify で過度なメモリ量の浪費を検出するだけでなく、アプリケーションのパフォーマンスとテストカバレッジを向上させることも可能です。これには、PurifyPlus に含まれている **PureCoverage** と **Quantify** を使用します。**PureCoverage** を使用すると、コードで未テストの箇所が報告されます。**Quantify** を使用すると、コードのパフォーマンスを低下させるボトルネックを検出できます。詳細については、57 ページの「**Rational PureCoverage** ファーストステップ」と 73 ページの「**Rational Quantify** ファーストステップ」を参照してください。

## Purify と .NET 管理コード: 基本手順

不要なオブジェクトへの参照が残っているためオブジェクトをガーベッジ収集できない場合、管理コードアプリケーションの実行中に消費するメモリは次第に大量になっていきます。Purify では、管理コードプログラムのメモリ使用量を測定し、これらの「メモリリーク」の原因になっているオブジェクトを的確に検出します。また、強制ガーベッジ収集によってコードのパフォーマンスが向上する箇所も指摘できます。

Purify を使用して管理コードのメモリ使用状況をプロファイルするには

- 1 Purify を使用して管理コード プログラムを実行します。
- 2 初期化完了後、スナップショットを撮ります。
- 3 リークしていると思われるコードを実行し、再度スナップショットを撮ります。
- 4 2つのスナップショットを比較して、メモリの問題の原因になっていると思われるメソッドを特定します。
- 5 これらのメソッドによって割り当てられ、リークしたオブジェクトを検出し、オブジェクトがガーベッジ収集されないようにしている参照を特定します。

ここでは、Microsoft Visual Studio .NET と統合された Purify の使用方法について説明します。Purify は、これ以外の方法で使用することもできます。ほかの使用方法については、以下を参照してください。

- 54 ページの「Purify の単独使用」
- 54 ページの「管理コード テスト環境への Purify の統合」

## Purify を使用して管理コード プログラムを実行する

Visual Studio .NET で初めて Purify を使用するときは、Visual Studio の [表示] メニューの [ツールバー] をポイントし、[Purify] をクリックして [Purify] ツールバーを表示します。ここでは、[Purify] ツールバーについて説明しますが、[Purify] メニューの対応するコマンドを使用することもできます。

Visual Studio .NET 内の管理コード プログラムを Purify するには、Visual Studio でプロジェクトを開き、[Purify] ツールバーの Purify 統合機能ボタンをクリックして、Purify 統合機能をオンにします。

Purify の統合機能をオンにするには、このボタンをクリックします。




Visual Studio のメニュー コマンドを使用して、プログラムをビルドして実行します。Purify で最も詳細なメモリ プロファイリング データを取得するには、プログラムをデバッグ データ付きでビルドします。

**詳細情報:** Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「プログラムの実行」を参照してください。

プログラムのランが進行するにしたがって、.NET ランタイム環境からのメモリ使用状況に関するメッセージがインターセプトされ表形式で報告されます。これらのメッセージに基づいて、プログラムによって各メソッドとオブジェクトに割り当てられたメモリ量が逐次追跡されます。

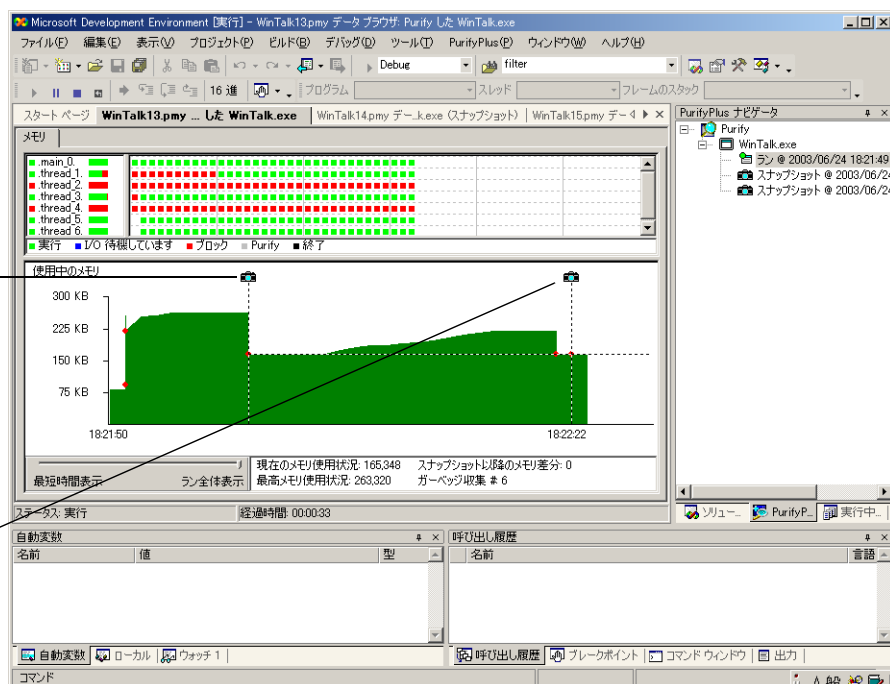
## メモリ使用状況のスナップショットを撮る


管理コードプログラムのメモリ リークにフォーカスして分析するには、プログラムの初期化完了後、[Purify] ツールバーの  をクリックして現在のメモリ使用状況のスナップショットを撮ります。このスナップショットは、プログラム実行時のメモリ使用状況を分析するためのベースラインです。

メモリをリークしていると思われるプログラム ルーチンを実行します。プログラムのランが進行するにしたがって、Purify の [データ ブラウザ] ウィンドウの [メモリ] タブに、プログラムのメモリ使用状況を示すグラフが表示されます。

プログラムの初期化完了後、最初のスナップショットを撮ります。

メモリ使用量が増加していることを確認し、再度スナップショットを撮ります。




グラフのメモリ使用状況の変化に注意します。メモリ使用量が著しく増加した場合、特に  をクリックして強制的にガーベッジ収集してもメモリ使用量が減少しないときなどは、問題が潜んでいる可能性があります。

プログラムの実行前後のメモリ使用状況を記録するため再度スナップショットを撮り、プログラムを終了します。

**詳細情報:** Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「スナップショットの作成」と「ガーベッジ収集」を参照してください。

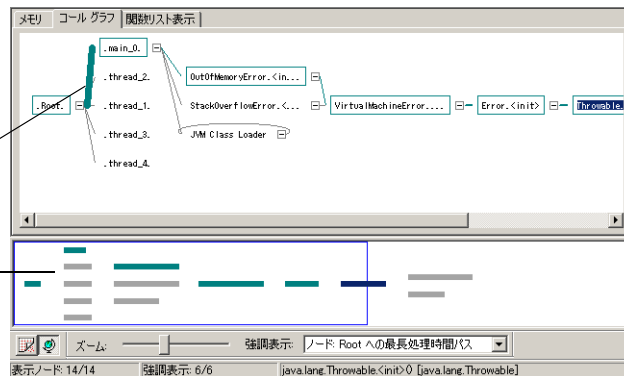
## スナップショットを比較して重要なコードにフォーカスする

[ナビゲータ] ウィンドウで 2 番目のスナップショット エントリを選択し、[Purify] ツールバーの  をクリックして、最初のスナップショットと比較します。

[コール グラフ] タブが表示され、最初のスナップショットと 2 番目のスナップショット間に最も多くのメモリを割り当てたメソッドが報告されます。

割り当てられているメモリ量は線の太さによって示されます。2 つのスナップショットの間で最もメモリ量に差があるパスが、最も太い線で表示されます。

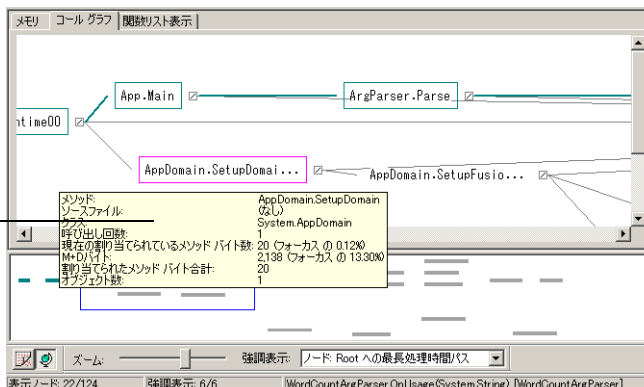
[コール グラフ概要] ウィンドウでは、[コール グラフ] のどの部分を表示しているのかを簡単に把握できます。



[コール グラフ] タブには、メソッド間の呼び出し関係も表示されます。このデータにより、不要なオブジェクトへの参照が残り、ガーベッジ収集が不可能になっているメソッドが検出できます。

分析するメソッドまたはパスの上にカーソルを移動します。ツールチップが表示され、そのメソッドのメモリに関連したデータが表示されます。

メモリ使用状況データは、[コール グラフ] タブで報告されます。



このデータにより、メモリを浪費しているメソッドとその下位メソッドにフォーカスして分析できます。

Purify でソース コードを表示するには、ソース コードが利用可能なメソッドをマウスの右ボタンでクリックし、ショートカット メニューの [ソース ファイル] をクリックします。

**詳細情報:** Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「スナップショットの差分」、「コール グラフ」、「ソース コード」、「表示」を参照してください。

**[関数リスト表示] タブを使用してリークを診断する**

[データ ブラウザ] ウィンドウの [関数リスト表示] タブには、同じデータがテキストで非階層表示されます。[関数リスト表示] タブでは、プログラム全体のデータを並べ替えて、メモリの消費量が最も多いメソッドを検出できます。

メモリ プロファイリングデータを並べ替えるには、列見出しをクリックします。

コール グラフ 関数リスト表示						
メソッド	呼び出し (回数)	呼び出し (パス)	現在 の呼び出し されているメソッド バイト数(差分)		クラス	
ctor	1	0	65,548		System.Xml.XmlShwType	
JIT Compiler	2,354	115	37,174		Runtime Internals	
XmlScanner.XmlScanner	2	0	16,412		System.Xml.XmlScanner	
String.GetStringForStringBuil...	121	4	15,566		System.String	
XmlAttributeTokenInfo.GetR...	56	0	9,900		System.Xml.XmlAttributeTokenInfo	
FileStream.Read	28	0	8,216		System.IO.FileStream	
XmlScanner.XmlScanner	1	0	8,206		System.Xml.XmlScanner	
TypeNameCache.TypeNameC...	1	0	7,364		System.Reflection.Cache.TypeNameCache	
String.Concat	63	0	5,344		System.String	
Type.GetTypeFromHandle	212	0	5,066		System.Type	
String.Substring	132	2	4,956		System.String	
Delegate.get_Method	4	0	4,176		System.Delegate	
Hashtable.HashTable	27	0	3,888		System.Collections.Hashtable	
String.Concat	56	0	3,196		System.String	
NameTable.GrowBuckets	6	0	3,132		System.Xml.NameTable	
Hashtable.expand	5	0	2,448		System.Collections.Hashtable	
ArrayList.ArrayList	48	1	2,400		System.Collections.ArrayList	
Enum.GetHashEntry	20	0	2,134		System.Enum	
StringBuilder.StringBuilder	14	1	1,644		System.Text.StringBuilder	
StringExpressionSet.Process...	15	0	1,394		System.Security.Linq.StringExpressionSet	
XMLUnitParseElementForOth...	7	0	1,376		System.Security.Linq.XMLUnit	
Tokenizer.Tokenizer	8	0	1,340		System.Security.Linq.Tokenizer	
ArgumentOutOfRangeException	21	1	1,176		System.ArgumentOutOfRangeException	

表示ノード: 2407/2407

**詳細情報:** Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「[関数リスト表示] タブ」を参照してください。

## [関数詳細] ウィンドウを使用してメソッドにフォーカスする

[コール グラフ] または [関数リスト表示] で任意のメソッドをダブルクリックすると、[関数詳細] ウィンドウが表示されます。このウィンドウには、選択したメソッド、その上位メソッド、その下位メソッドがメモリをどのように割り当てたかが報告されます。

メソッド: AppDomain.SetupDomain  
呼び出し回数: 1  
現在の割り当てられているメソッド バイト数: 20 (フォーカスの 0.12%)  
割り当てられたメソッド バイト合計: 20  
オブジェクト数: 1  
M+D バイト: 2,138 (フォーカスの 13.30%)  
平均 M バイト: 20  
最小 M バイト: 20  
最大 M バイト: 20  
クラス: System.AppDomain  
ソースファイル: (なし)  
非表示メソッド: (なし)

[ 上位関数 ] 列または [ 下位関数 ] 列にあるメソッドのいずれかをダブルクリックすると、そのメソッドのメモリデータが表示されます。

上位関数	現在の割り当てられているメソッド バイト数
Runtime00	2,138

下位関数	現在の割り当てられているメソッド バイト数
AppDomain...	2,118
AppDomain...	0
JIT Compil...	0

下位関数: 3      System.AppDomain.SetupDomain(value class System.LoaderOptimization)


**詳細情報:** Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「[関数詳細] ウィンドウ」を参照してください。

あるメソッドが割り当てたメモリ量が予想以上に多い場合はソース コードを調べます。別のメソッドや下位メソッドの参照しているオブジェクトがメモリのガーベッジ収集を妨げていることも考えられます。たとえば、下位メソッドに文字列配列の一部として静的変数が含まれていると、その配列全体のメモリがスコープ内にとどまり、プログラムの速度を低下させたりプログラムが応答しなくなる原因になります。

メモリの問題の原因になっていると思われるメソッドが見つかった場合は、メソッドのオブジェクトを調べます。Purify では、メソッドだけでなく、プログラム内のすべてのオブジェクトとそのメモリ使用状況に関するさまざまな情報が提供されます。

## 不要なオブジェクトの検索

プログラムで不要になったオブジェクトによりメモリがガーベッジ収集されず、その結果プログラムの速度が次第に低下していくことはよくあることです。Purify では、いろいろな形式でオブジェクトの総合メモリ データが報告されるので、この種の問題を簡単に追跡することができます。

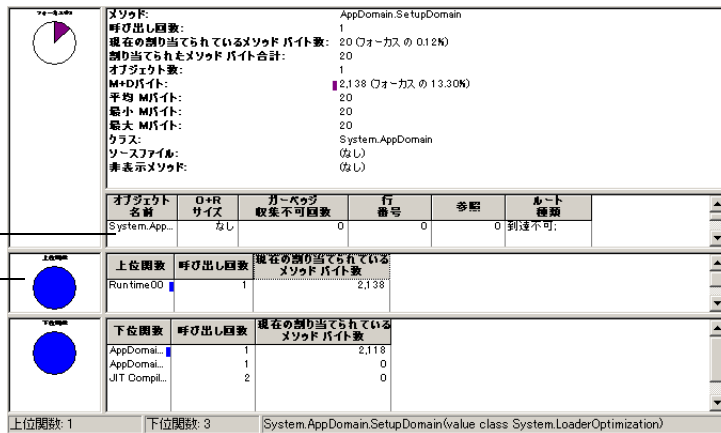
**メモ:** オブジェクト データを分析するには、スナップショット データを使用します。 をクリックすると作成される比較データには、オブジェクト データは含まれていません。

## 疑わしいメソッドからそのオブジェクトまでのデータを取得する

[関数詳細] ウィンドウには、メソッドに関する情報に加え、メソッドによって割り当てられたオブジェクトも一覧表示されます。列見出しをクリックすると、リスト内のオブジェクトを並べ替えることができます。

メソッドが現在割り当てられているオブジェクト。オブジェクトをダブルクリックすると、[オブジェクト詳細] ウィンドウにそのオブジェクトの総合メモリ データが表示されます。

スナップショット用の [関数詳細] ウィンドウには、メモリ割り当てを示す円グラフも表示されます。

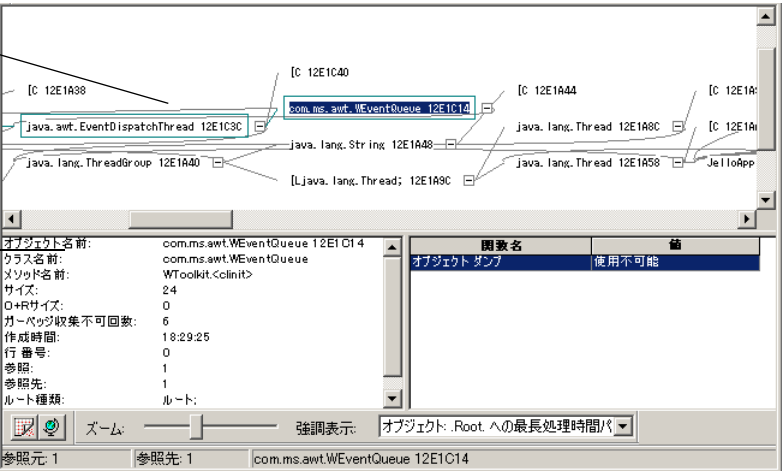


# オブジェクト詳細の分析

[関数詳細] ウィンドウでオブジェクトのいずれかをダブルクリックすると、[オブジェクト詳細] ウィンドウが表示されます。このウィンドウには、選択したオブジェクトのメモリ関連情報がすべて表示されるので、大量のメモリを保持しているオブジェクトを特定してその存続期間を決定できます。

オブジェクト参照グラフでは、現在のオブジェクトを参照しているオブジェクトと、現在のオブジェクトによって参照されているオブジェクトがそれぞれ表示されます。

サイズや作成時刻など、オブジェクト参照グラフで現在選択されているオブジェクトの詳細データ



# 割り当てられたすべてのオブジェクトを確認する

プログラムのトップレベルのオブジェクトを表示するには、メモリの問題があると思われるスナップショットの[データブラウザー]ウィンドウを開き、[オブジェクトリスト表示]タブをクリックします。

リストのアイテムを並べ替えるには、列見出しをクリックします。

プログラムで現在割り当てられているすべてのトップレベルオブジェクトのメモリデータ

ステータスバーには、選択された行番号とオブジェクトの合計数が表示されます。

メモリ   コールグラフ   関数リスト表示   オブジェクトリスト表示						
オブジェクト名前	メソッド名前	サイズ	O+R サイズ	ガーベッジ収集不可回数	行番号	
com.ms.ui.windowmanager.InputQueue...	SystemQueue.init	104	なし	9	9	(C)
com.ms.awt.WPanelPeer 12E1F2D	WToolkit.createPanel	56	なし	9	9	(C)
com.ms.awt.WSystemQueue 12E1A...	WToolkit.<init>	48	なし	9	9	(C)
com.ms.awt.WSystemQueue 12E1A...	WToolkit.<init>	48	なし	9	9	(C)
com.ms.ui.windowmanager.PaintOac...	Node.getVisRgn	40	なし	9	9	(C)
com.ms.ui.windowmanager.Message...	SystemQueue.init	24	なし	9	9	(C)
com.ms.ui.windowmanager.Message...	InputQueue.<init>	24	なし	9	9	(C)
java.awt.Insets 12E1F3D	WPanelPeer.initialize	24	なし	9	9	(C)
com.ms.ui.windowmanager.Message...	SystemQueue.init	24	なし	9	9	(C)
[Lcom.ms.ui.windowmanager.WUpdate; 12E1AE8	WSystemQueue.<init>	24	なし	9	9	(C)
java.lang.Object 12E1ACD	SharedWindowManager.<init>	12	なし	9	9	(C)
com.ms.ui.windowmanager.Message...	Node.getVisRgn	40	なし	9	9	(C)

オブジェクト: 7/13 | com.ms.ui.windowmanager | com.ms.ui.windowmanager.InputQueue.<init> (java.lang

スナップショットを撮ったときに割り当てられていたすべてのトップレベルオブジェクトが一覧表示されます。オブジェクトのリストには、オブジェクトのサイズ、オブジェクトの作成時刻、ガーベッジ収集ができなかった回数などいろいろな情報が表示されます。リストを並べ替えると、最も多くのメモリを保持しているオブジェクトと最も古いオブジェクトを簡単に検索できます。

リストのオブジェクト名をダブルクリックすると、そのオブジェクトの [オブジェクト詳細] ウィンドウが表示されます。

不要になったと思われるオブジェクトを検索する場合は、ソースコードを参照します。オブジェクトが実際に不要であることがわかった時点で、そのオブジェクトへのすべての参照を解放してガーベッジ収集できるようにコードを変更します。

**詳細情報:** Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「[関数詳細] ウィンドウ」、「[オブジェクト詳細] ウィンドウ」、「[オブジェクト リスト表示] タブ」を参照してください。

## Purify メモリ プロファイリング データの保存

Purify のデータを保存して後で分析したり、ほかのチーム メンバーと共有したり、報告書などに添付したりすることができます。Purify の管理コード データは次の形式で保存できます。

- Purify メモリ プロファイリング ファイル (.pmf): ラン、スナップショット、ほかのデータの場合と同じように、Purify で表示できるファイルです。
- ASCII テキスト ファイル (.txt): スクリプトで処理したり、スプレッドシートまたはワープロ プログラムで開いたりすることができます。

**詳細情報:** Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「データの保存」を参照してください。

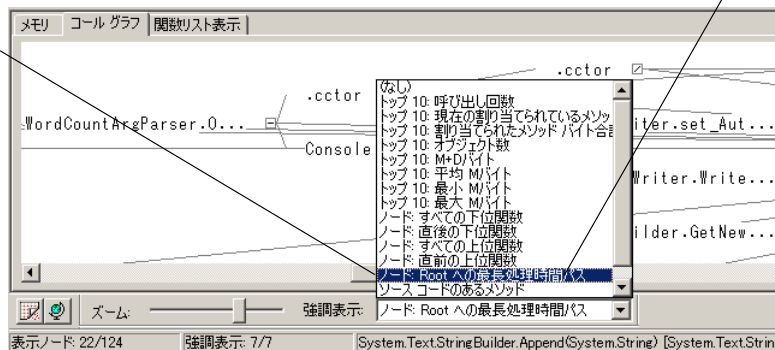
## Purify と .NET 管理コード: 高度な機能

### 同一の主要属性を持つメソッドの強調表示

特定のメモリ関連のデータまたは呼び出し関係を表示する場合は、該当するメソッドを [コール グラフ] で強調表示します。

たとえば、[ノード: Root への最長処理時間パス] をクリックすると、最も多くのメモリが割り当てられているパスで、選択したメソッドと .Root との間にあるすべてのメソッドが強調表示されます。

ここをクリックすると、強調表示リストが表示されます。



現在のデータにある 124 の関数のうち 22 関数が [コール グラフ] に表示されています。

.Root への最長処理時間パスにある 7 つの関数がすべて [コール グラフ] に表示されています。

**詳細情報:** Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「強調表示」を参照してください。

### データのフォーカス

Purify のフィルタ関連のコマンドを使用すると、選択したメソッド、またはあるクラス ファイル内のすべてのメソッドを Purify データで非表示にすることができます。または、サブツリー関連のコマンドを使用して、特定のメソッドとその下位メソッドにフォーカスしたり、Purify データから非表示にすることもできます。フィルタとサブツリー関連のコマンドは、[コール グラフ]、[関数リスト表示]、関数詳細のいずれかでメソッド名をマウスの右ボタンでクリックすると表示されるショートカット メニューにあります。

個々のメソッド、あるクラス ファイルのすべてのメソッド、サブツリー全体を非表示にしたり削除したりすることができます。

メソッドまたはサブツリーを非表示にすると、そのメモリは合計されてそれぞれの上位メソッドのデータに残ります。メソッドまたはサブツリーを削除すると、メモリは完全に破棄されます。



フィルタ マネージャを使用すると、その他のフィルタ オプションも利用できます。



選択したサブツリーにないメソッドをすべて削除するには、ショートカットメニューの [サブツリーにフォーカス] をクリックします。

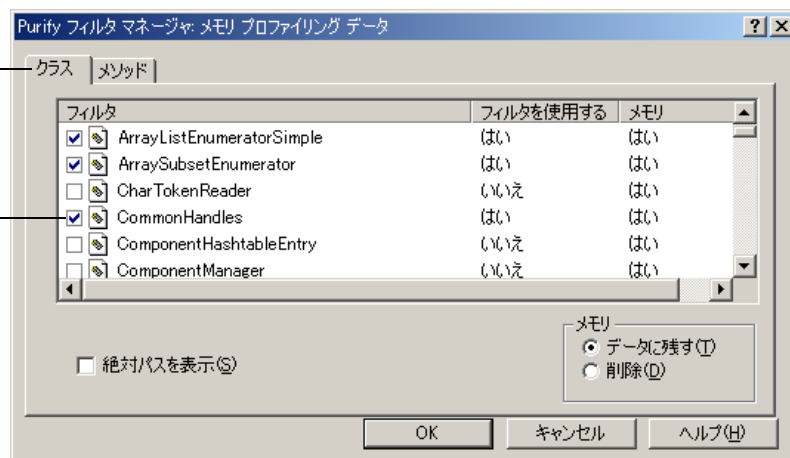
Purify には、すべてのフィルタとサブツリー関連のコマンドの操作を元に戻す機能があり、データをコマンド適用前の状態に戻すことができます。

[コール グラフ] には、サブツリーでの作業に便利なデータの展開と折りたたみ関連のコマンドもあります。しかし、フィルタとサブツリー関連のコマンドとは異なり、これらのコマンドは [コール グラフ] に表示されているデータ表示のみ影響します。データ自体は変更されません。

メニュー コマンドのほかに、フィルタ マネージャを使用して必要なデータを選択する方法もあります。

クラス ファイルまたはメソッドごとにデータをフィルタできます。

フィルタのオン/オフを切り替えるには、ここをクリックします。

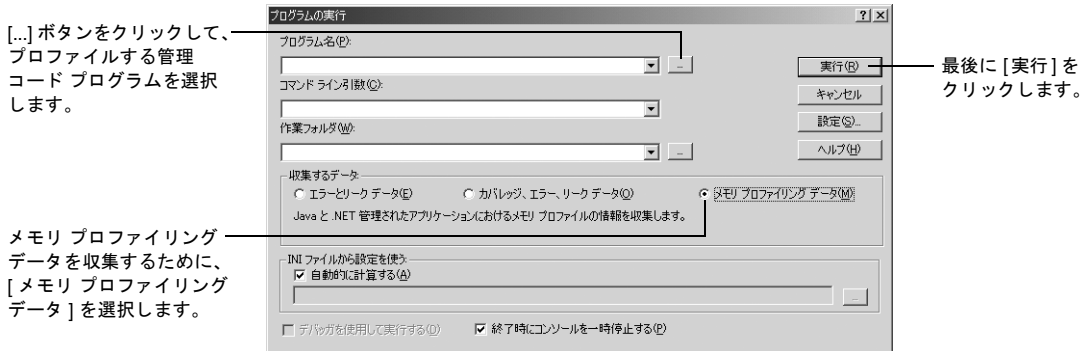


**詳細情報:** Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「データのフィルタ」と「サブツリー」を参照してください。


## Purify の単独使用

Microsoft Visual Studio .NET のリソースの一部のみが必要な場合は、Purify を単独で使用することもできます。Visual Studio との統合機能を使用する場合と同様、Purify の使いやすい単独使用ユーザー インターフェイスでも、メモリ プロファイリングを効率よく行うことができます。

単独使用の Purify を使用するには、[スタート] メニューから Purify を起動します。次に、表示される [ようこそ] ダイアログ ボックスで [実行] をクリックし、[プログラムの実行] ダイアログ ボックスを表示します。



コードがインストールメントされ、[データ ブラウザ] ウィンドウに診断結果が報告されます。

**詳細情報:** ツールバーのボタン、メニュー コマンド、ダイアログ ボックスの詳細については、 をクリックした後に該当アイテムをクリックします。

## 管理コード テスト環境への Purify の統合

Purify コマンドライン インターフェイスを使用すると、自動テスト環境でメモリ プロファイリング データを収集することができます。既存の makefile、バッチ ファイル、Perl スクリプトを変更し、Purify でプログラムを実行します。たとえば、管理コードプログラムを実行するテスト スクリプトの場合は、その管理コードプログラムを実行する行を次のように変更します。

```
Purify /SaveData /Net Exename.exe
```

このコマンドでは、管理コードプログラムを実行してメモリ プロファイリングデータを収集し、そのデータを .pmy ファイルに保存します。このファイルは、Purify インターフェイスで開いて分析したり、ほかのチーム メンバーと共有することができます。

データを .txt ファイルに保存する場合は、/SaveData オプションではなく /SaveTextData オプションを使用します。このデータを処理するスクリプトを作成して、プログラムのメモリ使用状況に関するレポートを生成することができます。たとえば、現在の夜間テストのデータと、以前の夜間テストのデータを比較するときに、メモリ関連のリグレーションを発生と同時に検出します。

自動データ収集を制御して、比較可能なデータを各テストから確実に生成するには、Purify API を使用します。次の「Purify API を使用して管理コードメモリ プロファイリングを制御する」を参照してください。

**詳細情報:** Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「コマンドライン」を参照してください。

## Purify API を使用して管理コード メモリ プロファイリングを制御する

Purify には、メモリ プロファイリング機能を強化する API 関数が搭載されています。

API は、自動テストを行う場合に特に便利です。API 関数を使用して、プロファイルされている部分のコードを判別し、プログラムの初期化動作を除外して特定のモジュールまたはルーチンにフォーカスすることができます。初期化完了後データを消去して、プログラムの実行と同時にデータ収集を継続し、プログラムの終了直前にデータを保存することもできます。これは、Purify ユーザー インターフェイスで 2 つのスナップショットを比較する場合と同じことです。

**詳細情報:** カバレッジ モニター用の関数を含むすべての Purify API 関数のリストを参照するには、Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「API 関数リスト」を参照してください。各関数の使用法の詳細については、Purify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「API 関数、使用法」を参照してください。



Purify を管理コードで使用するための基礎知識はここまでです。Purify の使用中に疑問が生じたときは、Purify のオンライン ヘルプを参照してください。オンライン ヘルプには、Purify の各機能について、より詳しい説明が記載されています。



# Rational PureCoverage

## ファースト ステップ

### PureCoverage: 主な機能

---

製品を出荷する前には、担当するコードのすべての行、関数、プロシージャ、メソッドが機能することを徹底的に確認する必要があります。

PureCoverage を使用すれば、このような作業を優位に行うことができます。PureCoverage では、テストの完全性を自動的に評価し、テストでカバーできていないコード部分を突き止めることができます。Visual C++、Visual Basic、Java、.NET 管理コードのプログラムは、プログラムを実行するときに、テスト カバレッジを簡単にモニターできます。品質管理エンジニアは、PureCoverage をテスト ルーチンに統合して、実行した各テストに対する包括的なカバレッジ レポートを自動的に生成できます。

PureCoverage を使用すると、次の作業を行うことができます。

- テストされたコードとテストされなかったコードのパーセンテージを、コードの実行直後に表示できます。
- 未テストまたは十分にテストされていない関数、プロシージャ、メソッドを特定できます。
- ソース コード内にある個々の未テスト行を検出できます。
- データ収集をカスタマイズして、効率を最大限高めることができます。
- 表示をカスタマイズして、必要な詳細にフォーカスできます。
- プログラムの複数ランのカバレッジ データをマージできます。
- カバレッジ データを保存して、ほかのチーム メンバーと共有したり、レポートを生成したりできます。
- Microsoft Visual Studio と Microsoft Visual Basic と統合した PureCoverage を使用して、開発環境内からコード カバレッジをモニターできます。

## プログラム内のすべてのコンポーネントのチェック

PureCoverage では、次のプログラムにあるすべてのコンポーネントのカバレッジを分析します。

- .exe ファイル、.dll ファイル、OLE/ActiveX コントロール、COM オブジェクト内の Visual C/C++ コード
- Visual Basic プロジェクトと P-Code の .exe ファイル、ネイティブ コードの .exe ファイル、.dll ファイル、OLE/ActiveX コントロール、COM オブジェクト
- Java アプレット、クラス ファイル、JAR ファイル、コンテナ プログラムによって起動されるコード
- Microsoft Visual Studio .NET で生成される .NET 管理コード .exe ファイル
- コンテナ プログラムから起動したコンポーネント (Microsoft Internet Explorer、Microsoft Transaction Server、jexegen を使用してビルドした実行可能ファイル、Jview.exe、Tstcon32.exe、Netscape Navigator、Microsoft Office 各アプリケーションなど)
- Microsoft Excel と Microsoft Word のプラグイン

関数とモジュールに関する情報は、Java メソッドとクラス ファイル、Visual Basic プロシージャとオブジェクト ライブラリにも適用できます。

## エンジニアリング サイクル全体での PureCoverage の使用

開発サイクルとテスト サイクルの初期段階から一貫して PureCoverage を使用することにより、公式テストと非公式テストの双方におけるテストの抜け穴を検出して、取り除くことができます。初期段階からすべてのコードをテストして、エラーを検出するため、それらのエラーを修正するための十分な時間が確保できます。製品の最終リリースまで、新しいコードまたは変更したコードをテストするたびに、継続して PureCoverage を使用します。

### 開発者のためのヒント

新しいルーチンを開発したばかりだとします。PureCoverage を使用すると、カバレッジ データを収集して、新しいコードに関する情報に簡単にフォーカスできます。チェックインする前に、すべてのコードをテストしたかどうかをテスト直後に確認できます。開発者は最低限の作業を行うだけで、PureCoverage でカバレッジ データを収集できます。

手動でコードをテストする場合は、PureCoverage を使用してテストをモニターし、PureCoverage のガイドに従ってテストを行ってください。PureCoverage には、テストしたコードの関数、プロシージャ、メソッドのパーセンテージがインタラクティブに表示されます。

PureCoverage は、Microsoft Visual Studio、Microsoft Visual Basic と自動的に統合されます。このため、これらの環境でコードを開発する場合は、作業過程を変更しないで PureCoverage を使用できます。

### 品質管理エンジニアのためのヒント

品質管理エンジニアは、PureCoverage を使用して、テスト中のプログラムへの修正変更が、テスト ルーチンですべてテストされているかどうかを調べることができます。テスト スクリプトに 1 行または 2 行のコードを追加すると、テストの実行時に PureCoverage をパッチ モードで自動的に実行できます。テスト ルーチンの効率に関するフィードバックがすぐに継続的に送られてくるので、テスト中のプログラム内にあるすべての修正変更を確実にテストすることができます。

**詳細情報：**詳細なリファレンス情報と PureCoverage の段階的な使用法については、PureCoverage のオンライン ヘルプを参照してください。まず、PureCoverage オンラインヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「PureCoverage ファースト ステップ」を参照してください。

## PureCoverage: 基本手順

---

PureCoverage を使用すると、次の簡単な手順で、すべてのコードを確実にテストすることができます。

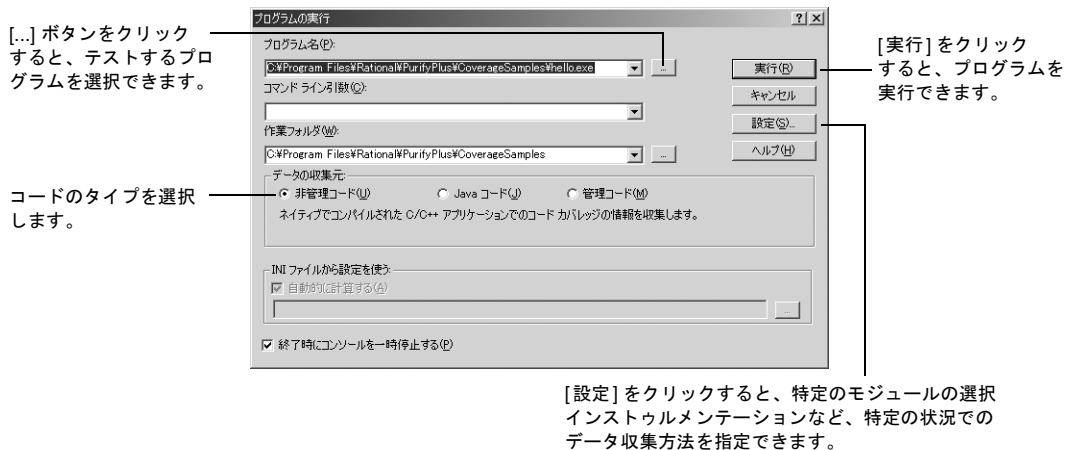
- 1 PureCoverage を使用してプログラムを実行します。
- 2 [カバレッジブラウザ] ウィンドウと [関数リスト] ウィンドウを使用して全体像を把握します。PureCoverage フィルタを使用して、注目する領域にフォーカスします。
- 3 [コメント付きソース コード] ウィンドウで、未テスト行を特定します。
- 4 テスト ランを変更して、未テストの行、条件、関数、プロシージャ、メソッドもテストでカバーされるようにします。
- 5 プログラムを再実行して、カバレッジが向上したことを確認します。カバレッジデータを保存して、ほかのチーム メンバーと共有できるようにします。

この章では、PureCoverage を単独使用デスクトップアプリケーションとして使用する方法について説明します。Microsoft Visual Studio または Microsoft Visual Basic と統合された PureCoverage を使用する場合は、または PureCoverage をテストルーチンに組み込む場合にも、同じ方法を適用できます。詳細については、66 ページの「PureCoverage と開発デスクトップの統合」と 71 ページの「テスト環境での PureCoverage の統合」を参照してください。

**メモ:** PureCoverage では、関数のカバレッジをモニターします。デバッグ行情報が使用可能な場合は、各行のカバレッジもモニターします。リリースモードでビルドされたプログラムの行レベルのデータが必要な場合は、デバッグ行情報を提供する必要があります。詳細については、PureCoverage オンラインヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「デバッグデータ」を参照してください。

## プログラムの実行

アプリケーションのコードカバレッジをモニターするには、[スタート] メニューから PureCoverage を起動します。次に、表示される [ようこそ] ダイアログボックスで [実行] をクリックし、[プログラムの実行] ダイアログボックスを表示します。

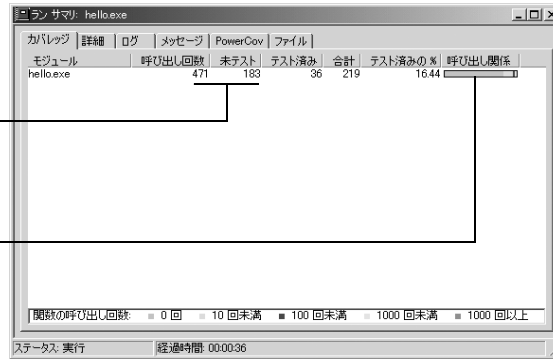


プログラムの実行が開始します。プログラム実行中は、テストされている行と関数に関する包括的な情報が収集されます。

プログラムの実行中に [ラン サマリ] ウィンドウが表示され、プログラム カバレッジの現在のステータスが表示されます。

プログラムの実行中は、  
[ラン サマリ] ウィンドウに、  
テスト済みと未テストの関数、  
プロシージャ、メソッドの  
数が表示されます。

関数、プロシージャ、  
メソッドの間での呼び  
出しの分布が、インジ  
ケータ内の色でわかります。



## 全体像の把握

[カバレッジブラウザ] ウィンドウと [関数リスト] ウィンドウでは、プログラムのカバレッジ ステータス全体が一目でわかります。

- 呼び出されていないすべての関数は、未テストとして報告されます。  
1 回以上呼び出された関数は、テスト済みとして報告されます。
- デバッグ行情報が **PureCoverage** で使用可能な場合は、未テスト コードとテスト済みコードの行数も報告されます。

この情報を使用すると、テストのホット スポットを簡単に特定できます。ホット スポットとは、テストされなかった主な領域です。

[カバレッジ ブラウザ] ウィンドウには、ソース ファイルに従って整理されたカバレッジ データが階層表示されます。

[モジュール表示] タブでは、モジュール内のファイルごとにデータが一覧表示されます。

[ファイル表示] タブでは、プログラム内のすべてのモジュール全体のファイルごとにデータが一覧表示されます。

[カバレッジ ブラウザ] ウィンドウには、関数、プロシージャ、メソッドに関するカバレッジ情報が階層形式で表示されます。

行に関するカバレッジ情報も階層形式で表示されます。

カバレッジ アイテム	呼び出し回数	未テスト 関数	テスト済み 関数	テスト済み 関数の%	未テスト 行	テスト済み 行	テスト済み 行の%
ラン @ 2003/05/30 18:23:41 <名無し>	142	21	10	32.26	1	15	93.75
C:\Program Files\Rational\WPU...	142	21	10	32.26	1	15	93.75
C:\Program Files\Rational\WPU...	2	0	2	100.00	1	15	93.75
hello.exe の (不明なファイル)	139	21	8	27.59			
WinMainCRTStartup	139	21	8	27.59			
atoi	0	未テスト					
atol	0	未テスト					
calloc	1		テスト済み				

[関数リスト] ウィンドウには、同じデータがテキストで非階層表示されます。  
[関数リスト] ウィンドウでは、プログラム全体のデータを並べ替え、プログラム全体でテスト済み部分が最も少ないコンポーネントを検出できます。


列見出しをクリックすると、プログラム全体のデータを並べ替えることができます。

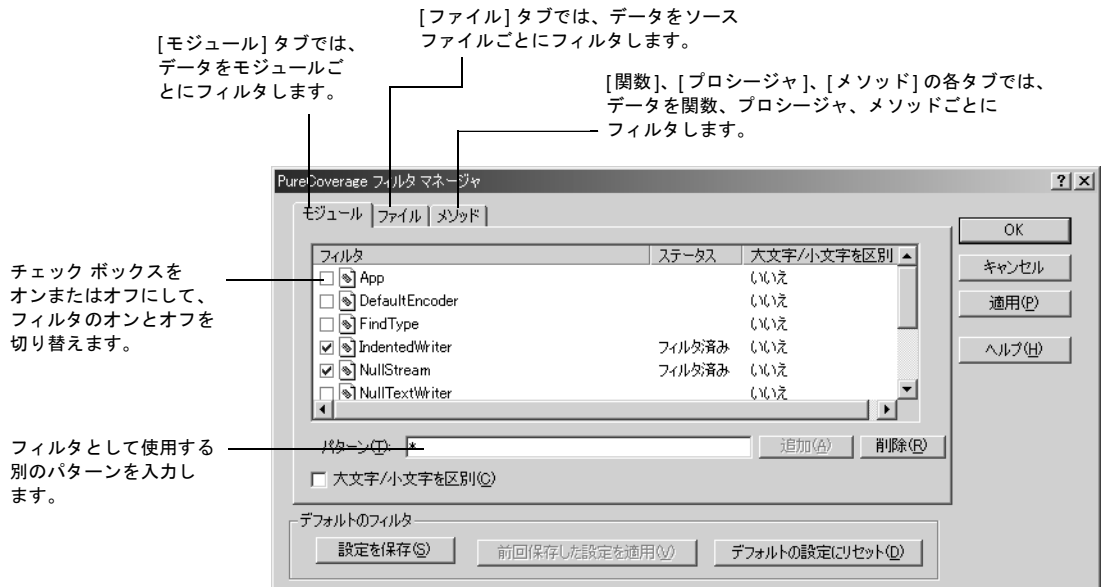
関数	呼び出し回数	合計 行	未テスト 行	テスト済み 行	テスト済み 行の%	モジュール	ソース ファイル
DisplayLocalTime	2	6	0	6	100.00	C:\Program Files\Rational\WPU...	C:\Program Files\Rational\WPU...
WinMain	1	10	1	9	90.00	C:\Program Files...	C:\Program Files\Rational\WPU...
atoi	0						hello.exe の (不明なファイル)
atol	0						hello.exe の (不明なファイル)
calloc	1						hello.exe の (不明なファイル)
exit	1						hello.exe の (不明なファイル)
free	1						hello.exe の (不明なファイル)
getenv	0						hello.exe の (不明なファイル)
localeconv	0						hello.exe の (不明なファイル)
malloc	37						hello.exe の (不明なファイル)
memcpy	0						hello.exe の (不明なファイル)
memmove	0						hello.exe の (不明なファイル)
memset	1						hello.exe の (不明なファイル)
realloc	0						hello.exe の (不明なファイル)
setlocale	0						hello.exe の (不明なファイル)

**詳細情報:** データ表示をカスタマイズする方法については、PureCoverage オンラインヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「カバレッジ データ ブラウザ」と「関数リスト」ウィンドウを参照してください。

## フィルタを使用したカバレッジ データのフォーカス

PureCoverage では、プログラム内のすべてのモジュールに関するカバレッジ情報が収集されます。ただし、デフォルトの設定では、収集されたすべてのデータが表示されるわけではありません。最も注目するカバレッジ情報を強調表示するには、デフォルトのフィルタ設定を適用して、プログラム内にある特定のシステムとサードパーティのコンポーネントのデータを非表示にします。

フィルタによって隠されたデータを表示する場合や、フィルタの設定を変更してほかのデータを表示する場合は、ツールバーの [フィルタ マネージャ] ボタン  をクリックして、[フィルタ マネージャ] ダイアログボックスを表示します。



**詳細情報:** PureCoverage オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「[フィルタ マネージャ] ダイアログ ボックス」を参照してください。

## 未テスト行の特定

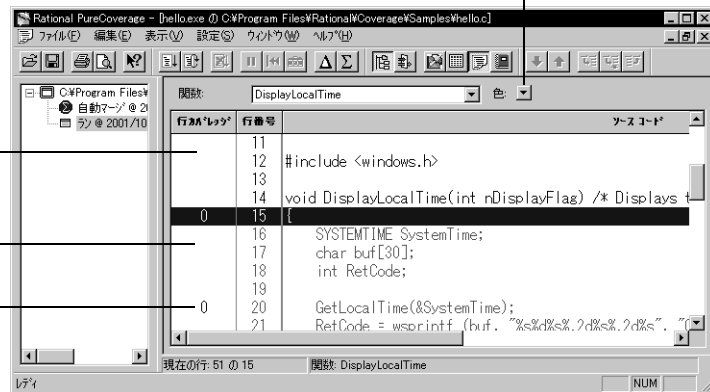
PureCoverage には、行ごとのカバレッジ データがデータ コメントとして、ソース ファイルのコピー内に表示されます。[カバレッジ ブラウザ] ウィンドウまたは[関数リスト] ウィンドウで関数、プロシージャ、メソッドのいずれかをダブルクリックすると、[コメント付きソース コード] ウィンドウにコードが表示されます。

このボタンをクリックすると、カバレッジの  
データ コメントの色を表示または変更できます。

[コメント付きソース  
コード] ウィンドウには、  
ソース コードのコピーが、  
行カバレッジに関する  
コメント付きで表示  
されます。

この行は 2 度テスト  
されました。

この行はテストされて  
いません。



デフォルト設定では、未テスト行は赤、テスト済み行は青、未使用行 (通常、コード内にアクティブな呼び出しがない関数、プロシージャ、メソッド内) は黒で表示されます。部分的にテスト済みの複数ブロック行は、ピンクで表示されます。これは、条件式などで可能な条件値を全部テストできなかった場合などに発生します。

Visual Studio の [クイック ウォッチ] ダイアログ ボックスまたは Visual Basic の [イミディエイト] ウィンドウを使用して、複数ブロック行を完全にテストすることができます。プログラムを実行した状態で、部分的にテスト済みの関数またはプロシージャの名前を入力して、テストする必要があるパラメータ値を指定します。

**詳細情報:** PureCoverage オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「[コメント付きソース コード] ウィンドウ」と「色、コメント付きソース コードの色の使用」を参照してください。[クイックウォッチ] ダイアログ ボックスまたは [イミディエイト] ウィンドウの詳細については、Visual Studio または Visual Basic のマニュアルを参照してください。

## テストランの変更

プログラムのテスト時にテストされなかったコードの箇所を検出できました。プログラムを非公式に実行している場合は、前回テストされなかったコードをテストする方法を検討します。テストルーチンでプログラムをテストしている場合は、テストスクリプトを追加または調整してカバレッジを改善します。

いずれの場合も、PureCoverage に表示された情報があるため、的確な作業が行えます。憶測で作業するのではなく、テストする必要があるコードの箇所が把握できているためです。

## プログラムの再実行

再度テストして、結果を確認します。新しいランのカバレッジデータだけでなく、自動マージデータも確認します。自動マージデータは、プログラムの新しいランと使用可能な前回のランのカバレッジデータから合成されます。

[ナビゲータ]ウィンドウには、マージしたデータが一覧表示されます。




カバレッジ アイテム	呼び出し回数	未テスト関数	テスト済み関数	テスト済み関数の%	未テスト行	テスト済み行	テスト済み行の%
ラン @ 2003/05/30 18:22:41 (3) [読み込み]	142	21	10	32.26	1	15	93.75
C:\Program Files\Rational\PureCoverage\bin\hello.exe	142	21	10	32.26	1	15	93.75
C:\Program Files\Rational\PureCoverage\bin\hello.exe (不明なファイル)	3	0	2	100.00	1	15	93.75
hello.exe (不明なファイル)	139	21	8	27.59			

特定のランのデータを手動でマージすることもできます。

**詳細情報:** PureCoverage オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「ランのマージ」を参照してください。一連のテストから自動的にデータをマージする方法については、71 ページの「テスト環境での PureCoverage の統合」を参照してください。

## カバレッジ データの保存

PureCoverage では、ほかのチーム メンバーと簡単に情報を共有できるため、テスト時間を節約することができます。データを保存して情報を共有するには、 ボタンをクリックします。

PureCoverage でサポートされるデータ形式は次の 2 つです。

- **PureCoverage データ ファイル (.cfy):** 後で PureCoverage で開いて、それ以降のプログラム ランを分析したり、比較することができます。または、PureCoverage を使用するほかのチーム メンバーと、.cfy ファイルを共有することもできます。
- **ASCII テキスト ファイル (.txt):** スプレッドシートまたはワープロ プログラムで使用します。また、電子メール メッセージや障害レポートで .txt ファイルを使用することで、テストのステータスを効率的に伝えることができます。

コマンドラインからデータを保存することもできます。インターフェイスを使用しないで夜間テストに PureCoverage を実行する場合は、この方法は不可欠です。

**詳細情報:** PureCoverage オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「データの保存」を参照してください。

## PureCoverage: 高度な機能

---

PureCoverage には、収集したカバレッジ データを最大限活用できる強力な機能が備わっています。たとえば、次のようなことができます。

- PureCoverage を開発デスクトップと統合できます。
- データ収集を微調整できます。
- 選択インストールメンテーションを使用して、プログラムのサブセットのデータを収集できます。
- 重要なプログラム領域にフォーカスできます。
- PureCoverage をテスト環境に統合できます。

この項では、これらの機能を使用して、コードをより効率的にモニターし、未テストのコード箇所をすばやく簡単にフォーカスできるようにします。

### PureCoverage と開発デスクトップの統合

PureCoverage を統合すると、使い慣れたツールを使用してコードを開発し、テストを行いながら、強力なカバレッジ データを活用することができます。PureCoverage は、Microsoft Visual Studio 6、Microsoft Visual Studio .NET、Microsoft Visual Basic、Visual Test、Robot、ClearQuest と統合できます。

インストール中に、PureCoverage のメニューとツールバーが Visual Studio 6 と Visual Basic に自動的に追加されるので、開発時には開発環境で作業しながらいつでもコードをモニターできます。Visual Studio .NET では、[表示] メニューの [ツールバー] をポイントし、[PureCoverage] をクリックすると、ツールバーが表示されます。



PureCoverage のツールバーにある [PureCoverage 統合機能] ボタンをクリックして、プログラムを実行します。

Visual Studio 6 または Visual Studio .NET 内で直接カバレッジ データを表示して作業します。

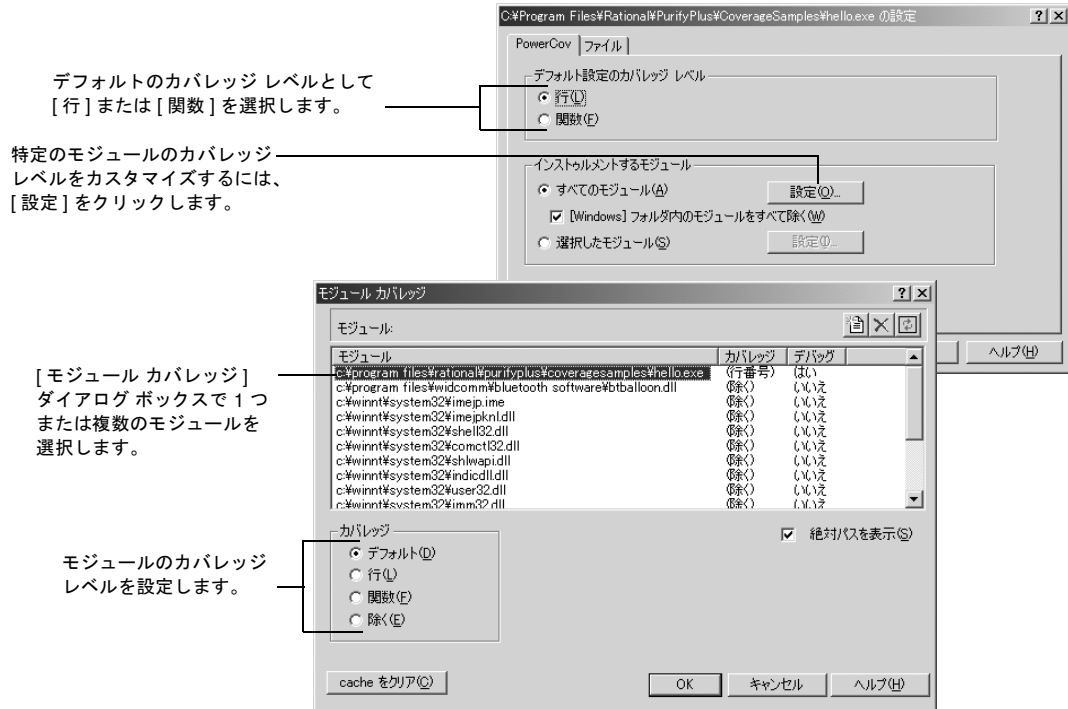
カバレッジ アイテム	呼び出し回数	未テストメソッド	テスト済みメソッド	テスト済みの%
ラン @ 2003/06/26 14:57	373	96	21	17.8
App	3	1	3	75.0
DefaultEncoder	41	1	2	66.6
FindType	2	35	2	5.4
IndentedWriter	2	9	1	10.0
NullStream	4	9	4	30.7
NullTextWriter	1	3	1	25.0
Runtime Internals	280	1	4	80.0
SyncTextWriter	39	35	3	78.8
UTF8Encoder	1	2	1	33.3

Visual Test または Robot が既にインストールされている場合は、Visual Test または Robot で作業しながら、プログラムに対するテスト スクリプトの実行と、プログラムのモニターを同時に行うことができます。ClearQuest と統合した場合は、未テスト コードが検出された時点で、PureCoverage で作業しながら、カバレッジ関連の障害を登録して、PureCoverage データ ファイル (.cfy) を添付することができます。

**詳細情報:** PureCoverage オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「統合」を参照してください。

## データ収集の微調整

PureCoverage の PowerCov のオプションを使用すると、開発段階やテスト段階でいつでも、プログラム内のモジュールに関して報告されるコード カバレッジのレベルを微調整できます。すべてのプログラムに適用されるデフォルトの設定を指定できます。また、現在のプログラムにのみ適用される設定を割り当てることもできます。



コード内の特定のモジュールにフォーカスするには、[PowerCov] タブのオプションを使用して、特定のモジュールのカバレッジ レベルに [行] を選択します。その他のモジュールのカバレッジ レベルに [関数] を選択すると、インストゥルメンテーションとランタイム パフォーマンスを向上させることができます。また、カバレッジから一部のモジュールを除外することもできます。

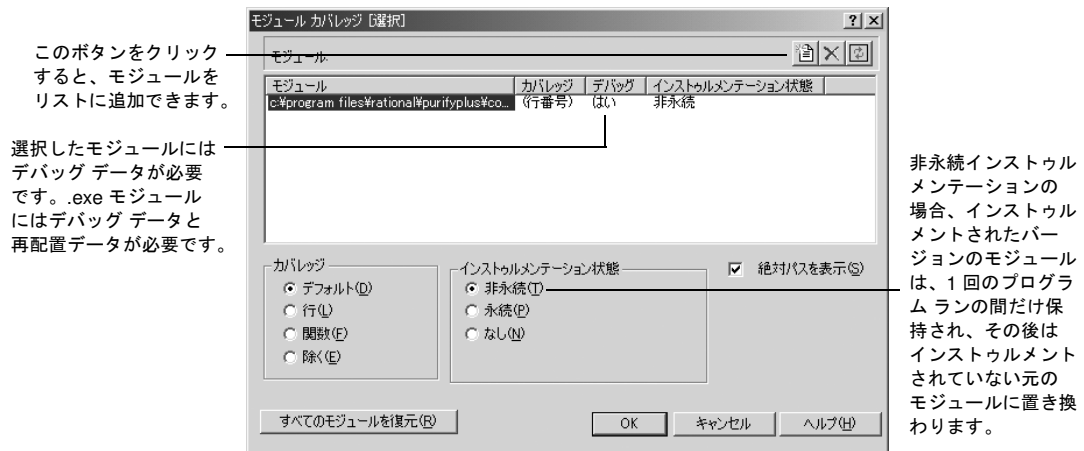
**詳細情報:** PureCoverage オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「設定、概要」と「カバレッジ レベル、概要」を参照してください。

## 選択インストールメンテーションの使用法

Visual C/C++ または Visual Basic のネイティブでコンパイルされたコードで作業する場合は、すべてのモジュールをインストールするのではなく、1 つまたは複数のモジュールまたは .dll ファイルをインストールするように選択できます。これにより、最も関心のあるコードにカバレッジデータを自動的にフォーカスすることができ、PureCoverage でのコード実行時の処理時間を節約できます。

たとえば、Microsoft Internet Information Server (IIS) によってロードされるプラグインアプリケーションで作業しているとします。この場合、すべての IIS をインストールしてプロファイルする必要はありません。使用中のプラグインのみをインストールしてから、IIS で通常どおり実行します。プラグインの実行中に PureCoverage でパフォーマンス データが収集され、プラグインが終了すると、このデータが表示されます。

プラグインをインストールするには、PureCoverage の [設定] メニューの [デフォルトの設定] をクリックし、[デフォルト設定] ダイアログ ボックスを表示します。次に、[インストールするモジュール] で [選択したモジュール] を選択します。[設定] をクリックして [モジュール カバレッジ] ダイアログ ボックスを開き、プラグインの名前を指定します。



通常どおりプラグインを実行します。プロファイリング データが収集され、表示されます。

**詳細情報:** PureCoverage オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「選択インストールメンテーション」を参照してください。

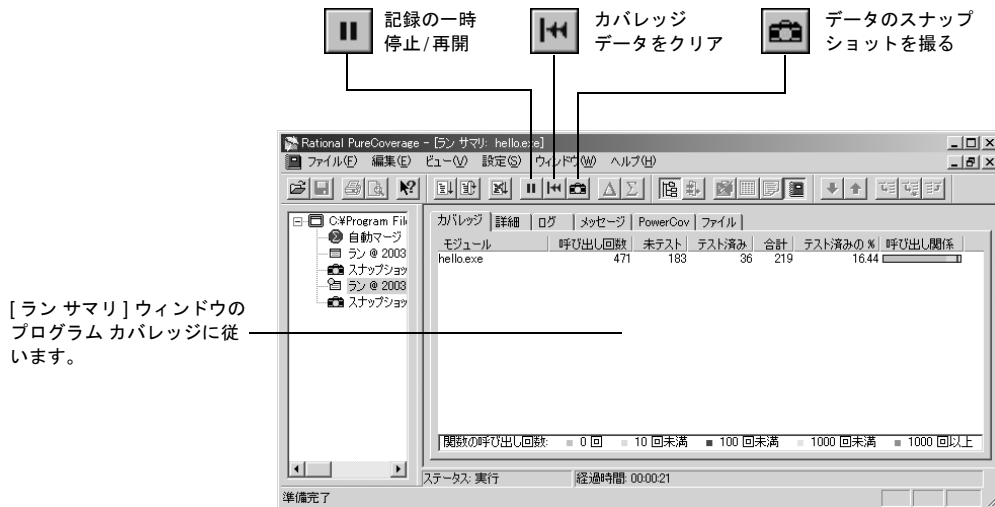
## 重要なプログラム領域へのフォーカス

PureCoverage では、プログラム全体またはプログラムの一部のカバレッジデータを取得することができます。特定の箇所のカバレッジ情報を取得するには、以下を使用します。

- インタラクティブなスナップショット
- PureCoverage API 関数

## インタラクティブなスナップショットの作成

PureCoverage では、プログラムのテストを実行しながら、各ルーチンのカバレッジデータのスナップショットを撮ることができます。



**詳細情報:** PureCoverage オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「スナップショット」と「[ラン サマリ] ウィンドウ」を参照してください。

## PureCoverage API 関数の使用法

PureCoverage のアプリケーション プログラミング インターフェイス (API) 関数を使用すると、カバレッジデータの収集に対する制御を強化できます。API 関数を使用すると、プログラムの実行中にいつでもデータ収集を開始/停止したり、データを保存したりできます。また、フォーカスする必要がある、プログラムの特定の領域に関するカバレッジデータだけを収集できます。

PureCoverage API 関数は、プログラム、Visual Studio の [ クイック ウォッチ ] ダイアログ ボックス、使用しているデバッガから呼び出すことができます。

**詳細情報:** PureCoverage オンライン ヘルプの [ トピックの検索 ] ウィンドウの [ キーワード ] タブでキーワード「API 関数、使用法」を参照してください。

## テスト環境での PureCoverage の統合

PureCoverage をテスト環境に統合すると、カバレッジを継続してモニターできます。たとえば、PureCoverage を既存の makefile、バッチ ファイル、Perl スクリプトから簡単に実行するには、次のコマンドを追加します。

```
Coverage /SaveTextData Exename.exe
```

これにより、PureCoverage でプログラムを実行できます。/SaveTextData オプションを使用すると、グラフィカルインターフェイスを使用しないで、テキストファイル形式でカバレッジデータを生成できます。テキスト ファイルの情報は、テスト結果レポートに組み込むことができます。

PureCoverage では、複数ランのカバレッジ データをマージすることも可能です。たとえば、あるプログラムに対して、毎回異なるデータを使用して一連の自動テストを実行していると仮定します。この場合、すべてのカバレッジ データを 1 つのファイルにマージするようにスクリプトを変更できます。それには、テスト スクリプトの冒頭に次の行を追加します。

```
del Exename_AutoMerge.cfy
```

これにより、既存のすべての自動マージ ファイルが削除されます。

次に、プログラムを実行する run コマンドを、次の行に置き換えます。

```
Coverage /SaveMergeData /SaveMergeTextData Exename.exe
```

このコマンドにより、プログラムのすべてのランから収集されたカバレッジ データがマージされ、PureCoverage データ ファイル Exename\_AutoMerge.cfy と、ASCII テキスト ファイル Exename\_AutoMerge.txt に保存されます。

Java、.NET 管理コード、Visual Basic のプログラマ: Java コードの場合は、コマンド ラインに /Java スイッチを含めます。.NET 管理コードと Visual Basic P-Code プログラムの場合は、コマンドラインに /Net スイッチを含めます。たとえば、Java クラス ファイルを実行するテスト スクリプトの場合は、その Java ファイルを実行する行を次のように変更します。

```
Coverage /SaveData /Java Java.exe Classname.class
```

管理コードと P-Code プログラムの場合は、コマンドは次のようになります。

```
Coverage /SaveData /Net Exename.exe
```



# Rational Quantify

## ファースト ステップ

### Quantify: 主な機能

---

「ソフトウェアは速ければ速いほどいい」、「電算リソースを最大限に利用して即応するプログラムがほしい」といったユーザーのニーズに応えられない、低パフォーマンスのアプリケーションでは、開発者が苦心して組み込んだ機能も利用価値が半減してしまいます。

どうしたらよいのでしょうか。

この問題の現実的な解決策は、体系的なパフォーマンス エンジニアリングによって、パフォーマンスのボトルネックを検出し、それを軽減するか、取り除くことです。プログラムが実行可能になった時点で、パフォーマンスのモニターを開始します。この時点で構造上の変更を行うのが最も簡単で、経済的です。出荷可能な状態になるまでパフォーマンスの調整を続けます。各パフォーマンスの向上のためにかかるコストと、調整によって得られる利益を比較してみてください。

それでは、パフォーマンス エンジニアリングに必要なデータを入手するにはどうしたらよいのでしょうか。

Quantify を使用すると、パフォーマンス エンジニアリングでの成功が約束されます。Quantify では、詳細で正確なパフォーマンス データが収集され、わかりやすいグラフや表に表示されるので、コードのどの部分に問題があるのかを正確に特定できます。Quantify を使用すると、事実上あらゆるプログラムの実行速度を向上させて、その結果を計測できます。

Quantify では、次のような一般に使用されているすべてのプログラミング言語で記述されたコードのパフォーマンスをプロファイルします。

- .exe ファイル、.dll ファイル、OLE/ActiveX コントロール、COM オブジェクト内の Visual C/C++ コード
- Visual Basic プロジェクトと P-Code の .exe ファイル、ネイティブ コードの .exe ファイル、.dll ファイル、OLE/ActiveX コントロール、COM オブジェクト
- Java アプレット、クラス ファイル、JAR ファイル、コンテナ プログラムによって起動されるコード

- .NET 管理コードアセンブリ、.exe ファイル、.dll ファイル、OLE/ActiveX コントロール、COM オブジェクト
- コンテナ プログラムから起動したコンポーネント (Microsoft Internet Explorer、Microsoft Transaction Server、jexegen を使用してビルドした実行可能ファイル、Jview.exe、Tstcon32.exe、Netscape Navigator、Microsoft Office 各アプリケーションなど)
- Microsoft Excel と Microsoft Word のプラグイン

Quantify では、ソース コードの有無にかかわらず、コードのすべてのコンポーネントをプロファイルできます。Visual C/C++ と Visual Basic で記述されたネイティブ コード アプリケーションについても、プロファイルするモジュールを的確に選択することができます。

Quantify は、Microsoft Visual Studio 6、Microsoft Visual Studio .NET、Microsoft Visual Basic と自動的に統合されます。このため、これらの環境でコードを開発する場合は、作業過程を変更しないで Quantify を使用できます。

この章では、Quantify を使用してパフォーマンスのボトルネックを検出する方法と、強力かつ柔軟なパフォーマンス エンジニアリング ツールである Quantify の機能について説明します。この章で説明する関数とモジュールに関する情報は、Java メソッドとクラス ファイル、Visual Basic プロシージャとオブジェクト ライブラリにも適用できます。

## Quantify: 基本手順

---

Quantify はプログラムとそのコンポーネントに関する詳細かつ正確なパフォーマンス データ セットを提供し、そのプログラムがどの処理で最も時間を費やすかを的確に示します。

プログラムのパフォーマンスを向上させるには

- 1 Quantify を使用してプログラムを実行し、パフォーマンス データを収集します。
- 2 Quantify データ ウィンドウを使用してパフォーマンス データを分析し、ボトルネックを検出します。
- 3 コードを変更して、ボトルネックを軽減させるか、取り除きます。
- 4 プログラムを再実行し、[ ランの比較 ] ツールを使用して、パフォーマンスが向上したことを確認します。

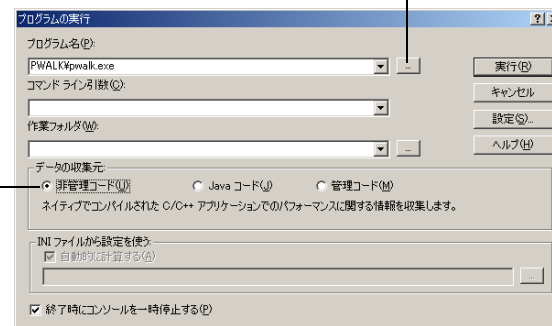
この章では、Quantify を単独使用デスクトップアプリケーションとして使用する方法について説明します。Microsoft Visual Studio または Microsoft Visual Basic と統合された Quantify を使用する場合は、または Quantify をテストルーチンに組み込む場合にも、同じ方法を適用できます。詳細については、85 ページの「Quantify と開発デスクトップの統合」と 92 ページの「テスト環境での Quantify の統合」を参照してください。

## プログラムの実行

プログラムのパフォーマンス データを収集するには、Windows の [スタート] メニューから Quantify を起動します。次に、表示される Quantify の [ようこそ] ダイアログ ボックスで [実行] をクリックし、[プログラムの実行] ダイアログ ボックスを開きます。

[...] ボタンをクリックすると、プロファイルするプログラムを選択できます。

コードのタイプを選択します。



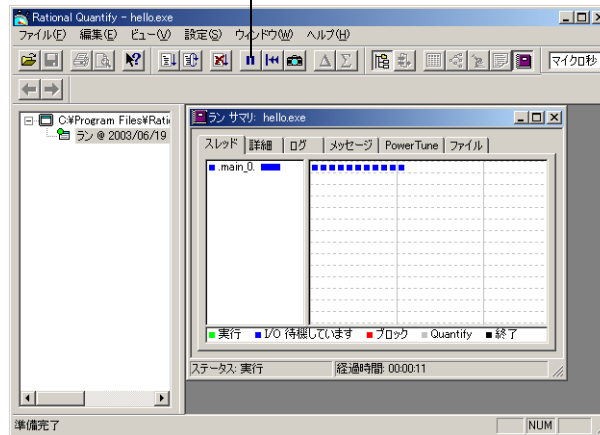
[実行] をクリックすると、プログラムを実行できます。

[設定] をクリックすると、特定のモジュールの選択、インストールメンテーションなど、特定の状況におけるデータ収集方法を指定できます。

Quantify では、関数のパフォーマンスをプロファイルします。デバッグ行情報が使用可能な場合は、各行のパフォーマンスもプロファイルします。リリースモードでビルドしたプログラムの行レベルのデータが必要な場合は、デバッグ行情報を提供する必要があります。詳細については、Quantify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「デバッグデータ」を参照してください。

プログラムの実行中に [ラン サマリ] ウィンドウが表示され、すべてのプログラム スレッドの現在のステータスが表示されます。

このボタンをクリックすると、特定のルーチンにフォーカスするためにプロファイルの一時停止/再開を切り替えることができます。



インストールしたコンポーネントはすべて保存されます。インストールしたコンポーネントを使用することで、プログラムの再実行時に時間を節約することができます。コンポーネントは、前回のラン以降に変更が追加された場合にのみ再度インストールされます。

プログラムの終了時には、収集したパフォーマンス データが表示されます。

**詳細情報:** Quantify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「選択インストールメンテーション」、「ラン」、「データの記録」を参照してください。

## パフォーマンス データの分析

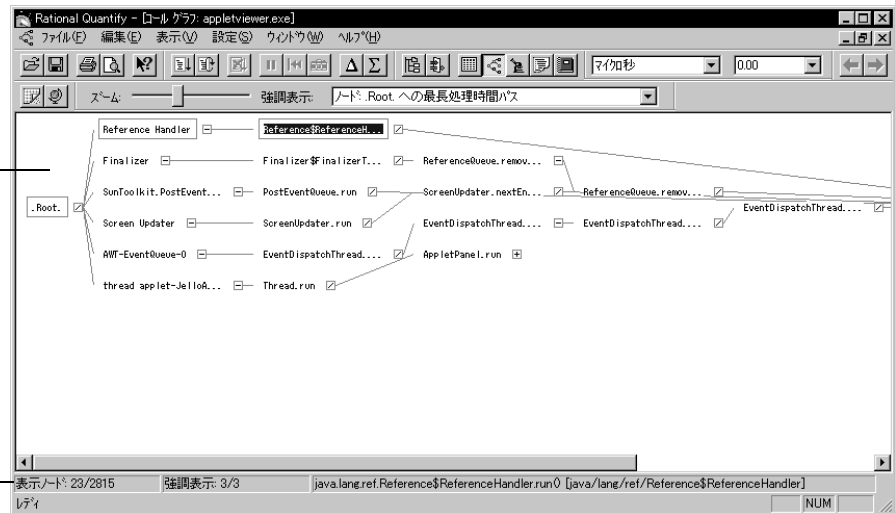
プログラムのパフォーマンスを向上させる第2の手順は、Quantify によって収集されたパフォーマンス データを分析することです。

## Quantify の [コール グラフ] ウィンドウの使用法

プログラムを終了すると、[コール グラフ] ウィンドウが表示されます。ウィンドウの初期設定の表示では、パフォーマンスの向上により最も影響を受ける領域である、コードの重要なコンポーネントがフォーカスされます。

Quantify の [コール グラフ] には、初期設定では、プログラム内で処理時間が長い 20 の関数が表示されます。

ランの処理時間の合計を表すルート ノードにより、最大 23 のノードが表示されます。



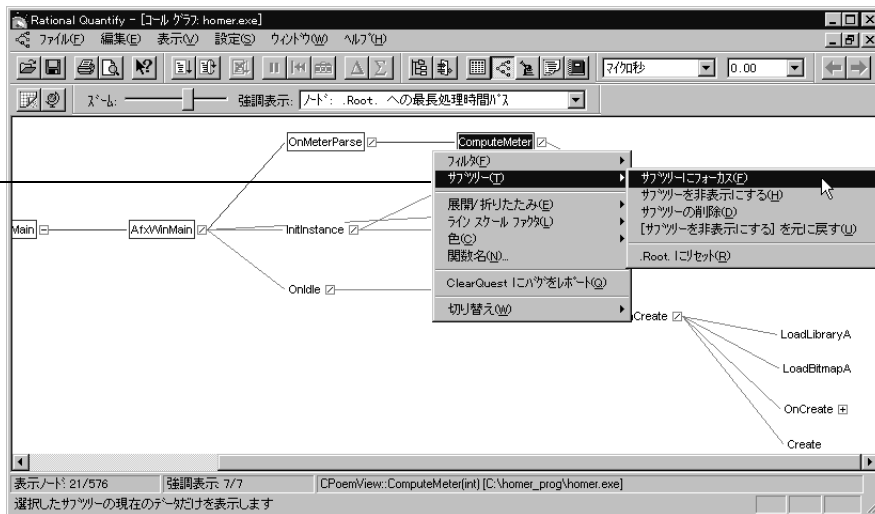
[コール グラフ] では、処理時間が最も長いパスが強調表示されます。また、パフォーマンス、呼び出し関係、ボトルネックの潜在的な原因など、さまざまな基準によって関数を強調表示することもできます。さらに、関数を追加表示したり、非表示にしたり、選択して移動したりして、呼び出し関係をより明確に表示することも可能です。

[コール グラフ] を使用して、処理時間が長すぎる関数を検索します。たとえば、このコードを記述したプログラムには、ComputeMeter 関数は初期 [コール グラフ] にまったく表示されないほど高速に処理されるべきであるということがわかっています。



処理時間が長いと疑われる関数を特定したら、その関数を隔離して、時間がかかっている箇所を確認します。

サブツリー関連の  
コマンドを使用  
すると、データの  
フォーカスを調整  
できます。



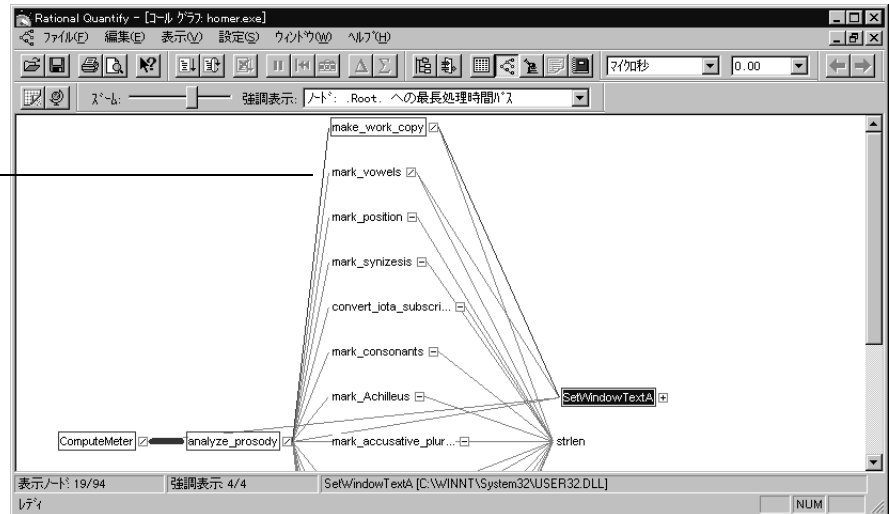
ComputeMeter 関数と下位関数のみが表示されるようにデータが調整されます。  
次に、ComputeMeter サブツリーを展開して、下位の状況を表示します。

展開 / 折りたたみ  
関連のコマンドを  
使用すると、プロ  
グラムの構造を  
確認できます。



処理時間が最も長い ComputeMeter サブツリー内のパスは、SetWindowTextA 関数であることがわかります。

線の太さから、パスの相対的な処理時間を判断できます。



このコードは、アルゴリズム開発時にフィードバックを行うために記述された関数であり、リリースされたアプリケーションで使用するものではありません。関数を削除すると、パフォーマンスが著しく向上します。

### 数値データ分析のための [関数リスト] ウィンドウの使用法

Quantify では、まず [コールグラフ] を使用して、現在のプログラムを分析します。次に、問題を絞り込む追加の方法を提供します。[関数リスト] ウィンドウを使用すると、数値パフォーマンスデータを表示して、並べ替えることができます。



[関数リスト] ボタンをクリックすると、数値データが表示されます。

この例では、[関数リスト] ウィンドウに、SetWindowTextA への不要な呼び出しにかかった正確な処理時間が表示されます。SetWindowTextA サブツリーのすべてのデータが表示されています。

[F+D 時間] には、関数とそのすべての下位関数の処理時間が含まれます。

これは、F+D 時間が最も長い関数の 1 つです。

関数	呼び出し回数	関数処理時間	F+D 時間	F 時間 (オーバーヘッドの%)	F+D 時間 (オーバーヘッドの%)	平均 F 時間
ComputeMeter	1	0.94	607,924.64	0.00	100.00	0.94
analyze_prosody	1	82.34	599,951.65	0.01	98.69	82.34
SetWindowTextA	2,929	210,938.51	296,491.81	34.70	48.77	72.02
strlen	40,554	288,857.84	288,857.84	47.52	47.52	7.12
make_work_copy	1	220.01	173,630.44	0.04	28.56	220.01
mark_vowels	1	127.89	165,474.34	0.02	27.22	127.89
mark_position	1	735.56	95,641.80	0.12	15.73	735.56
mark_synizesis	1	72.84	12,088.67	0.01	1.99	72.84
convert_jota_subscripts	1	263.46	10,745.26	0.04	1.77	263.46
mark_consonants	1	230.33	10,399.87	0.04	1.71	230.33
mark_Achilleus	1	57.78	10,259.37	0.01	1.69	57.78
mark_accusative_plurals	1	71.91	10,243.30	0.01	1.68	71.91
mark_finalvowels	1	141.18	10,209.56	0.02	1.68	141.18
mark_a_e	1	71.07	10,192.27	0.01	1.68	71.07
find_diphthongs	1	86.30	10,162.29	0.01	1.67	86.30

パーセンテージを参照すると、SetWindowTextA がサブツリーの処理時間の合計のほぼ 50% を占めていることがわかります。この関数は、プログラムの現在のバージョンでは用途がないため、パフォーマンスのボトルネックを発生させる最も一般的な原因の 1 つである、不要な処理の明確な例として挙げられます。

### インタラクティブな「what-if」分析の実行

Quantify では、プログラムの現在のパフォーマンスを分析する以外に、プロジェクトのパフォーマンスを予測することができます。

前に示した例では、[コール グラフ] で [SetWindowTextA] を右クリックしてから、SetWindowTextA サブツリーを削除することができます。表示されたデータからサブツリーの時間が削除され、残りのデータが再度計算されます。これによって、サブツリーを除いたプログラムのパフォーマンスを正確に確認できます。

ComputeMeter  
サブツリーの処理  
時間は、変更前は  
600,000 マイクロ秒  
でしたが、  
約 311,000 マイクロ  
秒に短縮されました。

関数	呼び出し 回数	関数 処理時間	F+D 時間	F 時間 (オーバーの%)	F+D 時間 (オーバーの%)	平均 F 時間
ComputeMeter	1	0.94	311,432.84	0.00	100.00	0.94
analyze_prosody	1	82.34	303,945.32	0.03	97.60	82.34
strlen	40,554	288,857.84	288,857.84	92.75	92.75	7.12
mark_position	1	735.56	95,641.80	0.24	30.71	735.56
make_work_copy	1	220.01	27,748.88	0.07	8.91	220.01
mark_vowels	1	127.89	16,443.10	0.04	5.28	127.89
mark_synizesis	1	72.84	12,088.67	0.02	3.88	72.84
convert_jota_subscripts	1	263.46	10,745.26	0.08	3.45	263.46
mark_consonants	1	230.33	10,399.87	0.07	3.34	230.33
mark_Achilleus	1	57.78	10,259.37	0.02	3.29	57.78
mark_accusative_plurals	1	71.91	10,243.30	0.02	3.29	71.91
mark_finalvowels	1	141.18	10,209.56	0.05	3.28	141.18
mark_a_e	1	71.07	10,192.27	0.02	3.27	71.07
find_diphthongs	1	86.30	10,162.29	0.03	3.26	86.30
mark_dubiousvowels	1	63.20	10,106.56	0.02	3.25	63.20

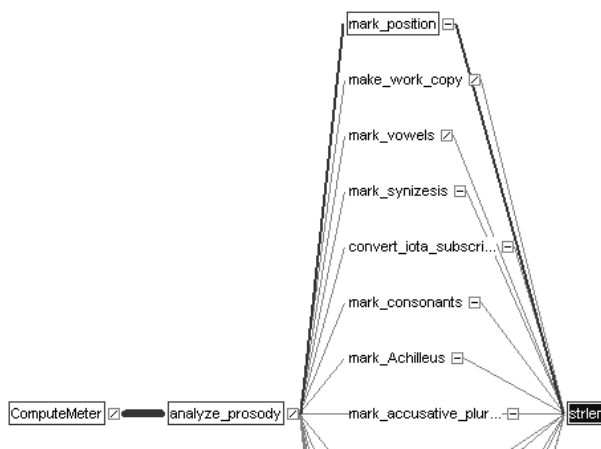
## [関数詳細] ウィンドウの使用法

[関数詳細] ウィンドウには、個々の関数の観点からパフォーマンス データを表示できます。



[関数詳細] ボタンをクリックすると、特定の関数のデータが表示されます。

この例では、strlen 関数が、[関数リスト] と [コール グラフ] の両方に表示されています。[関数リスト] を参照すると、プログラムの実行中に、この関数が 40,000 回以上呼び出されたことがわかります。[コール グラフ] を参照すると、プログラムのこの部分にある処理時間の長いすべての関数が、strlen を呼び出していることが確認できます。



コードのこの部分は、テキスト行を文字列として処理します。これらの関数は、複雑なルールの集合を順序どおりに各行に適用して、パターンを識別します。ただし、strlen がそれほど何度も呼び出される場合は、パフォーマンスに問題があると考えられます。

列見出しをクリックすると、リストを並べ替えることができます。

strlen は、サブツリー内で処理時間が最も長い単一の関数です。

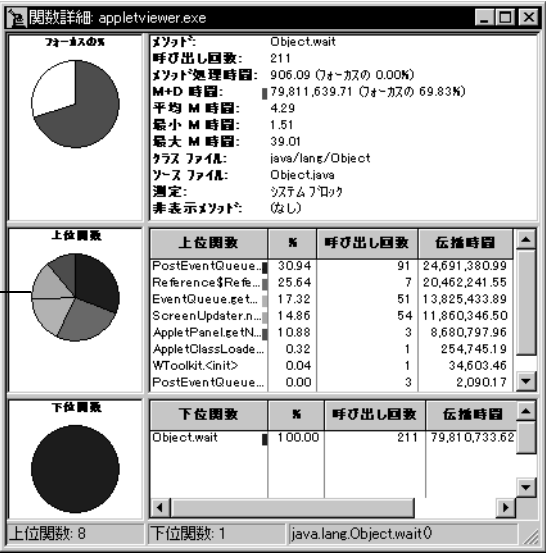
関数リスト: homer.exe						
関数	呼び出し回数	関数処理時間	F+D 時間	F 時間 (オーバーヘッドの%)	F+D 時間 (オーバーヘッドの%)	平均 F 時間
strlen	40,554	288,857.84	288,857.84	92.75	92.75	7.12
tolower	1,463	1,161.33	1,161.33	0.37	0.37	0.79
mark_position	1	735.56	95,641.80	0.24	30.71	735.56
ExtTextOutA	2	507.00	562.46	0.16	0.18	253.50

SetWindowTextA がデータから削除されたので、strlen は単独でサブツリーの処理時間の合計の約 92% を使用します。

[関数詳細] ウィンドウを開くと、その他の関数からの strlen への呼び出しに関する特定の情報が数値やグラフ形式で表示され、異なる角度からデータを確認することができます。

関数の詳細データ

[上位関数] または [下位関数] 円グラフの一部をダブルクリックすると、その関数のデータが表示されます。



関数への (上位関数からの) 呼び出しに関するデータ

関数からの (下位関数への) 呼び出しに関するデータ

このデータを分析すると、ほとんどの関数が、ほぼ同じ回数 strlen を呼び出していることがわかります。呼び出しの状況を正確に把握するには、ソースコードを参照します。

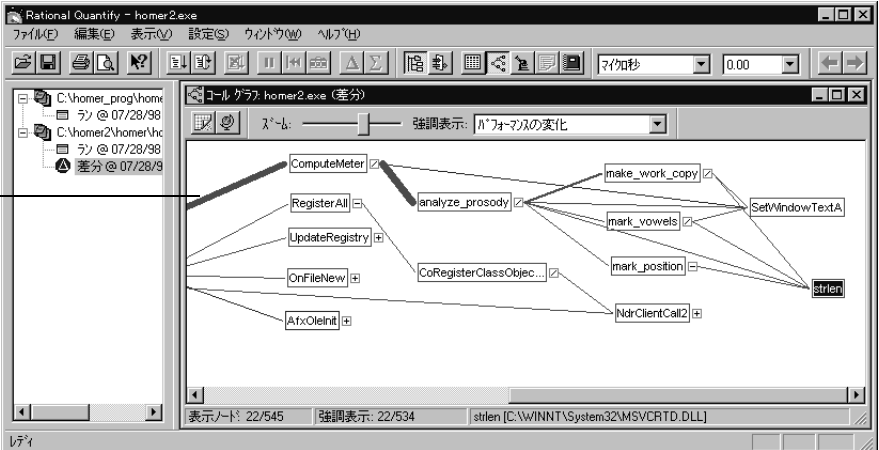


すべての strlen 呼び出しを取り除いて、プログラムを再実行したとします。最初のランと新しいランを比較して、パフォーマンスの変化を確認します。



[ランの比較] ボタンをクリックすると、パフォーマンスの変化が確認できます。

[コール グラフ] の差分ウィンドウでは、パフォーマンスが向上したパスと関数が緑色で強調表示されます。



[コール グラフ] の差分ウィンドウでは、ComputeMeter、strlen、SetWindowTextA は、パフォーマンスの向上を表す緑色で強調表示されます。差分の関数リストを開き、数値を比較します。

差分の関数リストでは、パフォーマンスの向上は負の値で表されます。

ComputeMeter の処理時間の合計は、約 12,000 マイクロ秒になり、595,000 マイクロ秒以上パフォーマンスが向上しています。

関数リスト: homer2.exe (差分)							
関数	F+D 時間 (差分)	F+D 時間 (前)	F+D 時間 (後)	関数処理 時間 (差分)	F 時間 (前)	F 時間 (後)	呼び出し (差分)
analyze_prosody	-596,939.23	3,012.42	599,951.65	-12.14	82.34	70.20	0
OnMeterParse	-595,664.54	12,673.44	608,337.98	0.00	0.12	0.12	0
ComputeMeter	-595,661.23	12,263.41	607,924.64	-0.01	0.94	0.93	0
SetWindowTextA	-308,965.79	326.41	307,292.20	-7,457.42	7,461.99	4.57	-2,927
strlen	-288,850.26	7.58	288,857.84	-288,850.26	288,857.84	7.58	-40,553
make_work_copy	-172,845.47	784.97	173,630.44	-42.95	220.01	177.06	0
mark_vowels	-165,339.74	134.59	165,474.34	-30.81	127.89	97.08	0
OnIdle	-105,621.61	601,279.77	706,901.38	261.38	78,222.31	78,483.70	25,586
OnIdle	-96,932.31	616,172.30	713,104.61	2,095.13	259.52	2,354.65	25,586
mark_position	-95,016.94	624.86	95,641.80	-110.70	735.56	624.86	0
InitInstance	-41,186.37	722,108.38	763,294.75	0.00	0.70	0.70	0
UpdateRegistry	-34,700.28	84,866.06	119,566.34	-16.44	600.66	584.41	0
ExtractIconA	-29,318.56	3,014.65	32,333.21	3.33	102.33	105.66	0
PrivateExtractIconsW	-28,728.05	30,655.72	59,383.77	824.09	6,646.13	7,470.22	0
AssertValid	-25,272.32	126,539.00	151,811.32	-47.17	268.25	221.08	-1,216

Quantify データ ファイル (.qfy) としてデータを保存すると、後でデータを分析したり、ほかの Quantify ユーザーとデータを共有したりできます。タブ区切り付きのテキスト ファイル (.txt) としてデータを保存すると、テスト スクリプトや Microsoft Excel などのアプリケーションで使用できます。また、[関数リスト] ウィンドウのデータをコピーし、Excel シートに貼り付けて使用することもできます。

## Quantify: 高度な機能

---

Quantify には、パフォーマンス データを最大限活用できる強力な機能が備わっています。たとえば、次のようなことができます。

- Quantify を開発デスクトップと統合できます。
- インストールメンテーションとプロファイルを行う特定のモジュールを選択できます。
- データの記録をインタラクティブに制御できます。
- 同一の主要属性を持つ関数を強調表示できます。
- 重要なデータにフォーカスできます。
- データ収集を微調整できます。
- Quantify をテスト環境に統合できます。

この項では、これらの機能を使用して、コードの重要な部分をより効率的にプロファイルし、ボトルネックを絞り込めるようにします。

### Quantify と開発デスクトップの統合

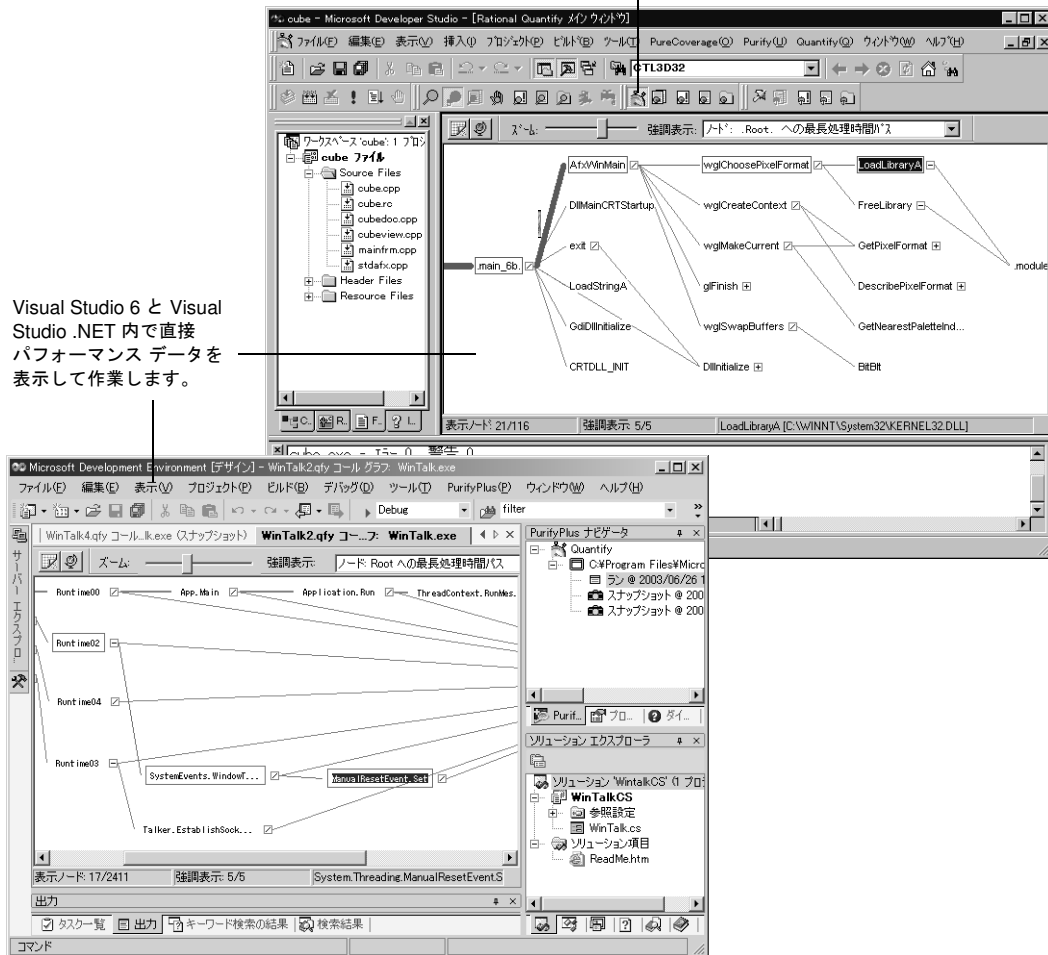
Quantify を、Microsoft Visual Studio、Microsoft Visual Basic、Visual Test、Robot、ClearQuest などと統合すると、使い慣れたツールを使用してコードを開発しながら、強力なパフォーマンス プロファイリングを行うことができます。

インストール中に Quantify のメニューとツールバーが Visual Studio 6 と Visual Basic に自動的に追加されるので、開発時には開発環境で作業しながらいつでもコードをプロファイルできます。Visual Studio .NET で初めて Quantify を使用するときは、Visual Studio の [表示] メニューの [ツールバー] をポイントし、[Quantify] をクリックして Quantify のツールバーを表示します。



Quantify のツールバーにある [Quantify 統合機能] ボタンをクリックして、プログラムを実行します。

Visual Studio 6 と Visual Studio .NET 内で直接パフォーマンス データを表示して作業します。



Visual Test または Robot が既にインストールされている場合は、Visual Test または Robot で作業しながら、プログラムに対するテスト スクリプトの実行と、プログラムのプロファイルを同時に行うことができます。ClearQuest と統合した場合は、実行速度が低下しているコードが検出された時点で、Quantify で作業しながら、パフォーマンス関連の障害を登録して、Quantify データ ファイル (.qfy) を添付することができます。

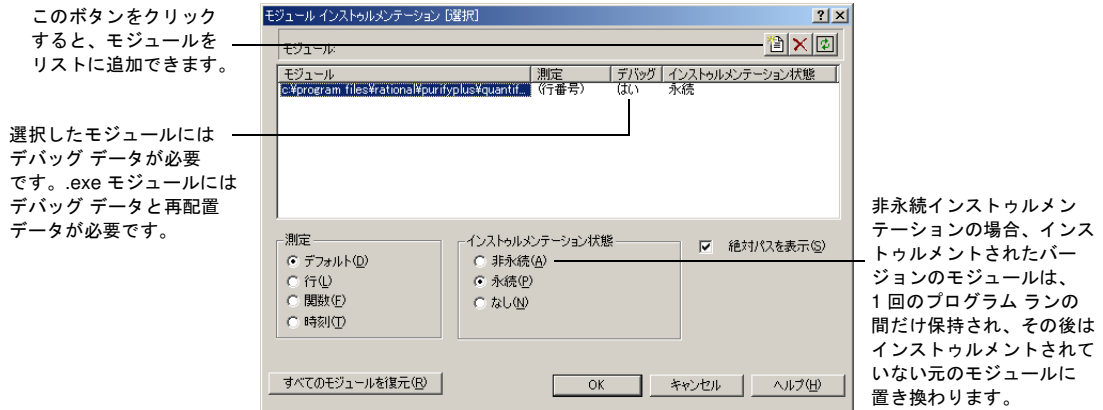
**詳細情報:** Quantify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「統合」を参照してください。

## 選択インストールメンテーションの使用法

Visual C/C++ または Visual Basic のネイティブでコンパイルされたコードで作業する場合は、すべてのモジュールをインストールするのではなく、1 つまたは複数のモジュールまたは .dll ファイルをインストールするように選択できます。これにより、最も関心のあるコードにプロファイリングデータを自動的にフォーカスすることができ、Quantify でのコード実行時の処理時間を節約できます。

たとえば、Microsoft Internet Information Server (IIS) によってロードされるプラグインアプリケーションで作業しているとします。この場合、すべての IIS をインストールしてプロファイルする必要はありません。使用中のプラグインのみをインストールしてから、IIS で通常どおり実行します。プラグインの実行中に Quantify でパフォーマンス データが収集され、プラグインが終了すると、このデータが表示されます。

プラグインをインストールするには、Quantify の [設定] メニューの [デフォルトの設定] をクリックし、[デフォルト設定] ダイアログ ボックスを表示します。次に、[インストールするモジュール] で [選択したモジュール] を選択します。[設定] をクリックして [モジュール インストールメンテーション] ダイアログ ボックスを開き、プラグインの名前を指定します。



通常どおりプラグインを実行します。プロファイリング データが収集され、表示されます。

**詳細情報:** Quantify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「選択インストールメンテーション」を参照してください。

## インタラクティブなデータの記録の制御

プログラム実行時に、スレッドとファイバのパフォーマンスをモニターし、[ラン サマリ] ウィンドウを使用してランに関する一般情報を表示することができます。



データ記録ツールを使用すると、プログラム全体のデータまたはプログラムの一部のデータを収集できるので、必要なパフォーマンス データのみを入手できます。たとえば、いつでも記録を中止して、その時点までに収集したデータをクリアし、記録を再開できます。また、現在のデータのスナップショットを撮り、ステージごとにパフォーマンスを検証できます。

Quantify のデータ記録の API 関数をコードに組み込むことで、プログラム内から自動的に記録を開始/停止したり、データをクリアしたり、スナップショットを撮ることもできます。

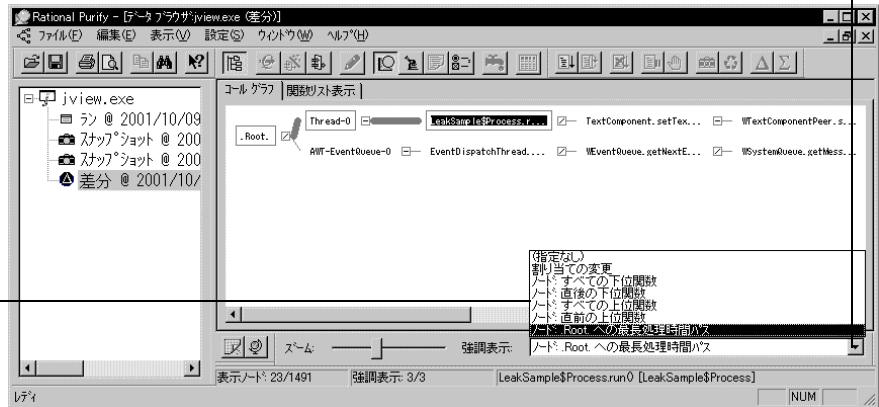
**詳細情報:** Quantify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「スレッド」、「データの記録」、「API 関数」を参照してください。

## 同一の主要属性を持つ関数の強調表示

特定のパフォーマンス データまたは呼び出し関係を表示する場合は、該当する関数を [ コール グラフ ] で強調表示します。

ここをクリックすると、強調表示リストが表示されます。

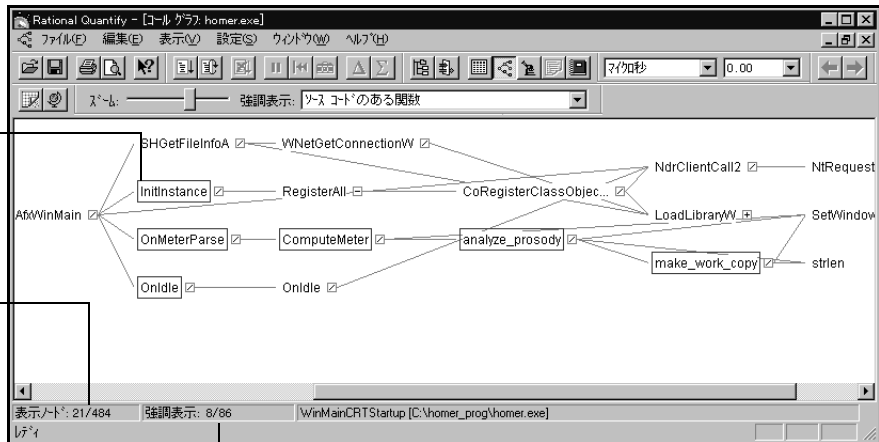
たとえば、[ ソースコードのある関数 ] を選択すると、コメント付きソースコードがある関数が強調表示されます。



ソースコードのある関数は、長方形で囲まれます。

現在のデータにある484の関数のうち21の関数が [ コールグラフ ] に表示されています。

ソースコードのある86の関数のうち8つの関数が [ コールグラフ ] に表示されています。



**詳細情報:** Quantify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「強調表示」を参照してください。

## データのフォーカス

Quantify のフィルタ関連のコマンドを使用して、選択した関数またはモジュール内のすべての関数を現在のデータから削除できます。または、サブツリー関連のコマンドを使用して、特定の関数とその下位関数にフォーカスしたり、それらを現在のデータから非表示にすることもできます。これには、[ コール グラフ ]、[ 関数リスト ]、[ 関数詳細 ] のいずれかのウィンドウ内で関数をマウスの右ボタンでクリックします。

個々のメソッド、あるクラス ファイルのすべてのメソッド、サブツリー全体を非表示にしたり削除したりすることができます。

メソッドまたはサブツリーを非表示にすると、そのメモリは合計されてそれぞれの上位メソッドのデータに残ります。メソッドまたはサブツリーを削除すると、メモリは完全に破棄されます。

フィルタ(F)	メソッド LeakSample\$Process.run() を非表示にする(H)
サブツリー(T)	クラス ファイル LeakSample\$Process を非表示にする(U)
展開/折りたたみ(E)	メソッド LeakSample\$Process.run() を削除する(D)
ライン スケール ファクタ(L)	クラス ファイル LeakSample\$Process を削除する(E)
色(C)	直前のフィルタ操作を元に戻す(U)
メソッド名(N)...	フィルタ マネージャ(M)...
ソース ファイル(S)	
データ フラウザ(B) Ctrl+B	

フィルタ マネージャを使用すると、その他のフィルタ オプションも利用できます。

フィルタ(F)	サブツリーにフォーカス(F)
サブツリー(T)	サブツリーを非表示にする(H)
展開/折りたたみ(E)	サブツリーの削除(D)
ライン スケール ファクタ(L)	[サブツリーを非表示にする] を元に戻す(U)
色(C)	.Root. にリセット(R)
メソッド名(N)...	
ソース ファイル(S)	
データ フラウザ(B) Ctrl+B	

選択したサブツリーにないメソッドをすべて削除するには、ショートカットメニューの [サブツリーにフォーカス] をクリックします。

Quantify には、すべてのフィルタとサブツリー関連のコマンドの操作を元に戻す機能があり、簡単にデータをコマンド実行前の状態に戻すことができます。

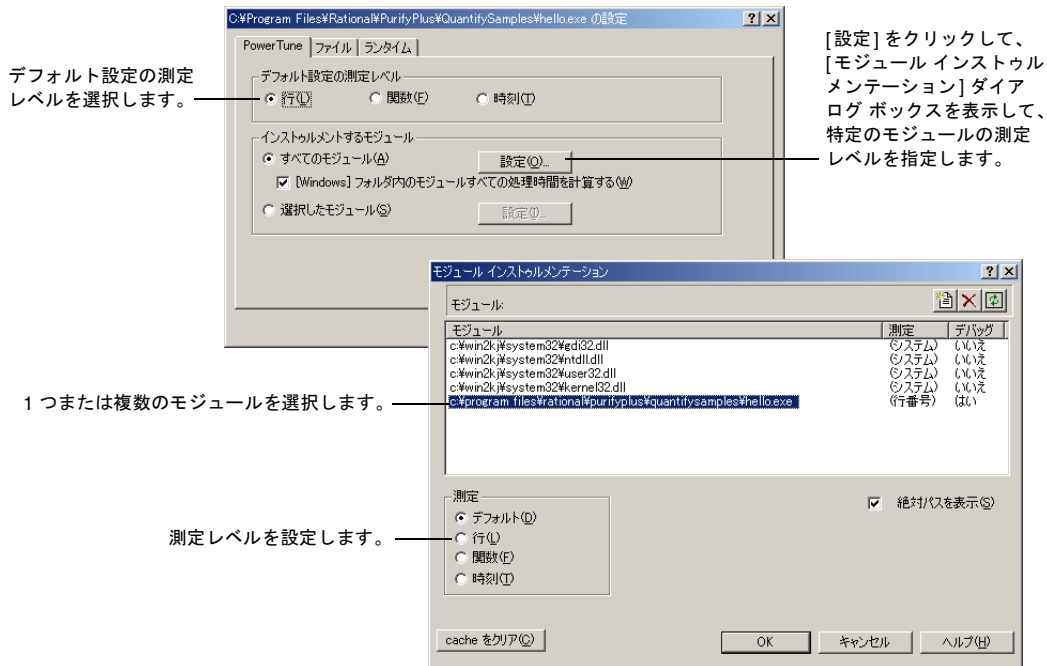
[ コール グラフ ] ウィンドウには、サブツリーでの作業に使用する展開と折りたたみ関連のコマンドもあります。ただし、これらのコマンドは、フィルタとサブツリー関連のコマンドとは異なり、[ コール グラフ ] 表示にのみ影響します。現在のデータ自体は変更されません。

**詳細情報:** Quantify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「データのフィルタ」と「サブツリー」を参照してください。

## データ収集の微調整

Quantify の [PowerTune] のオプションを使用して、Quantify でプログラムのパフォーマンスを測定する方法を指定できます。デフォルト設定の測定レベルは、ほとんどの状況に該当するレベルに基づいていますが、PowerTune を使用すると、特定のモジュールの測定方法を制御できます。

この方法が有益なのは、プロファイル中のランタイム パフォーマンスが著しく向上するためです。たとえば、デフォルト設定の測定レベルとして [時刻] を選択してから、現在分析している特定のモジュールについて [行] を選択します。



Quantify では次の各レベルでパフォーマンスを測定します。

- **行:** このレベルでは、ランの実行中に各行の実行回数をカウントして、1 回の実行に必要なサイクル数に基づいてパフォーマンス データを計算します。行レベルには、デバッグ行情報が必要です。最も精度が高く詳細なデータが収集されますが、データ収集に最も時間がかかります。
- **関数:** このレベルでは、行レベルと同じ精度でデータが収集されますが、データは行レベルほど詳細ではありません。各行のパフォーマンスを個別に確認する必要はないが、関数の正確な繰り返しデータが必要な場合は、このレベルでデータを収集すると有効です。
- **時刻:** Quantify では、各関数の開始/終了時にタイマを起動/停止することで、処理時間が計算される関数のデータを収集します。報告されるデータは現在のランについては正確ですが、マイクロプロセッサの状態やメモリ効果の影響を受けます。ただし、処理時間データの収集については、オーバーヘッドはほとんど発生しません。

**詳細情報:** Quantify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「測定タイプ」を参照してください。

## テスト環境での Quantify の統合

Quantify をテスト環境に統合することで、夜間テストでパフォーマンスの変化を検出するツールとして使用できます。これにより、不正なパフォーマンスが発生するとすぐにそこに注意を向けられます。

Quantify を既存の makefile、バッチ ファイル、Perl スクリプトから簡単に実行するには、次のコマンドを追加します。

```
Quantify /SaveData Exename.exe
```

これにより、Quantify でプログラムを実行できます。/SaveData オプションを使用すると、Quantify グラフィカル インターフェイスにプログラムの前回のランを表示、比較するように、パフォーマンス データを生成できます。

/SelectModuleList オプションを使用して、テストをフォーカスすることもできます。87 ページの「選択インストールメンテーションの使用法」を参照してください。

**Java、.NET 管理コード、Visual Basic のプログラマ:** Java コードの場合は、コマンドラインに /Java スイッチを含めます。管理コードと Visual Basic P-Code プログラムの場合は、コマンドラインに /Net スイッチを含めます。たとえば、Java クラス ファイルを実行するテスト スクリプトの場合は、その Java ファイルを実行する行を次のように変更します。

```
Quantify /SaveData /Java Java.exe Classname.class
```

管理コードと P-Code プログラムの場合は、コマンドは次のようになります。

Quantify /SaveData /Net Exename.exe

**詳細情報:** 詳細とその他のコマンドライン オプションについては、Quantify オンライン ヘルプの [トピックの検索] ウィンドウの [キーワード] タブでキーワード「コマンドライン」と「スクリプト」を参照してください。



ユーザー定義のコードで Quantify を使用してみましょう。Quantify の使用中に疑問が生じたときは、Quantify のオンライン ヘルプを参照してください。オンライン ヘルプには、Quantify の各機能について、より詳しい説明が記載されています。

---



# 索引

## A

ABW エラー (Purify、C/C++) 16

API 関数

PureCoverage 70

Purify、C/C++ 25

Purify、Java 41

Purify、管理コード 55

Quantify 88

ASCII テキスト ファイル (.txt)

PureCoverage 66, 71

Purify、C/C++ 21

Purify、Java 40

Purify、管理コード 55

Quantify 85

## C

C/C++ コード

Purify する 9

カバレッジのモニター 60

パフォーマンスのプロファイル 73, 75

.cfy ファイル

PureCoverage 66

Purify、カバレッジ データ 21

ClearQuest の統合

PureCoverage 67

Purify、C/C++ 25

Quantify 86

## D

dispose() メソッド (Purify、Java) 28

## F

F+D (関数 + 下位関数) 時間 (Quantify) 80

## J

Java (PureCoverage)

コマンドラインからの実行 71

対応言語 57

Java (Purify)

Java コードを Purify する 29

オブジェクトの分析 35–37

メモリ使用状況グラフ 31

メモリ プロファイリング データのフィルタ 39

メモリ プロファイリング データの保存 38

メモリ リーク 27, 29

Java (Quantify)

コマンドラインからの実行 92

対応言語 73

/Java オプション

PureCoverage 71

Purify 40

Quantify 92

## L

L+D (行 + 下位関数) 時間 (Quantify) 83

## M

Microsoft Visual Studio 6 の統合

PureCoverage 66

Purify 9

Quantify 85

Microsoft Visual Studio .NET の統合

PureCoverage 67

Purify 44

Quantify 85

## N

/Net オプション  
  PureCoverage 71  
  Purify 54  
  Quantify 92  
.NET 管理コード  
  「管理コード」を参照

## P

.pcy ファイル (Purify、C/C++) 21  
.pft ファイル (Purify、C/C++) 16  
.pfy ファイル (Purify、C/C++) 21  
.pmy ファイル  
  Purify、Java 38  
  Purify、管理コード 51  
[PowerCheck] タブ (Purify、C/C++) 21  
[PowerCov] タブのオプション (PureCoverage) 68  
PowerTune (Quantify) 91  
PureCoverage  
  PurifyPlus のツール 1  
  開発者のためのヒント 2  
  使用法 59  
  品質管理エンジニアのためのヒント 3  
Purify  
  PurifyPlus のツール 1  
  開発者のためのヒント 2  
  使用法 (C/C++) 9  
  使用法 (Java) 29  
  使用法 (管理コード) 44  
  品質管理エンジニアのためのヒント 3  
Purify C/C++ エラー メッセージの非表示  
  「フィルタ」を参照  
PurifyPlus、概説 1  
Purify する  
  C/C++ コード 9  
  Java コード 29  
  管理コード 44  
Purify 単独使用インターフェイス (C/C++) 23  
Purify 単独使用インターフェイス (管理コード) 54  
Purify データ ファイル  
  C/C++ 21  
  Java 38  
  管理コード 51  
Purify の埋め込み解除 (C/C++) 23

## Q

Quantify  
  PurifyPlus のツール 1  
  開発者のためのヒント 2  
  使用法 74  
  品質管理エンジニアのためのヒント 3

## R

Rational ClearQuest の統合  
  PureCoverage 67  
  Purify 25  
  Quantify 86  
Rational PureCoverage  
  PurifyPlus のツール 1  
  開発者のためのヒント 2  
  使用法 59  
  品質管理エンジニアのためのヒント 3  
Rational PureCoverage の使用法の戦略 58  
Rational Purify  
  PurifyPlus のツール 1  
  開発者のためのヒント 2  
  使用法 (C/C++) 9  
  使用法 (Java) 29  
  使用法 (管理コード) 44  
  品質管理エンジニアのためのヒント 3  
Rational PurifyPlus、概説 1  
Rational Quantify  
  PurifyPlus のツール 1  
  開発者のためのヒント 2  
  使用法 74  
  品質管理エンジニアのためのヒント 3  
Rational Robot の統合  
  PureCoverage 67, 72  
  Purify、C/C++ 25–26  
  Quantify 86  
Rational Visual Test の統合  
  PureCoverage 67, 72  
  Purify、C/C++ 25–26  
  Quantify 86  
Rational 技術サポート、連絡先 5  
Robot の統合  
  PureCoverage 67, 72  
  Purify、C/C++ 25–26  
  Quantify 86

## S

/Save\* オプション

- PureCoverage 71
- Purify、C/C++ 25
- Purify、Java 40
- Purify、管理コード 55
- Quantify 92

## V

Visual Basic

- 統合 (PureCoverage) 66
- 統合 (Quantify) 85

Visual C/C++、プログラムの実行

- PureCoverage 60
- Purify 9
- Quantify 75

Visual Studio 6 の統合

- PureCoverage 66
- Purify、C/C++ 9
- Quantify 85

Visual Studio .NET の統合

- PureCoverage 67
- Purify 44
- Quantify 85

Visual Test の統合

- PureCoverage 67, 72
- Purify、C/C++ 25–26
- Quantify 86

## W

what-if、Quantify コール グラフ 80

## い

色

- コメント付きソース コード (PureCoverage) 64
- コメント付きソース コード  
(Purify、カバレッジデータ) 19
- コール グラフ (Quantify) 84

インストールメンテーション

- 概説 (PureCoverage) 60
- 概説 (Purify、C/C++) 10
- カスタマイズ (PureCoverage) 68
- カスタマイズ (Purify、C/C++) 22
- 選択 (PureCoverage) 69
- 選択 (Quantify) 87
- デフォルト設定のレベル (Purify、C/C++) 21
- インタラクティブなスナップショット  
(PureCoverage) 70

## う

ウィンドウとタブ

- オブジェクト詳細 (Purify、Java) 36
- オブジェクト詳細 (Purify、管理コード) 50
- オブジェクト リスト表示 (Purify、Java) 37
- オブジェクト リスト表示  
(Purify、管理コード) 50
- カバレッジ ブラウザ (PureCoverage) 61
- 関数詳細 (Purify、Java) 34, 35
- 関数詳細 (Purify、管理コード) 48, 49
- 関数詳細 (Quantify) 82
- 関数リスト (PureCoverage) 62
- 関数リスト (Quantify) 80, 84
- 関数リスト表示 (Purify、Java) 33
- 関数リスト表示 (Purify カバレッジデータ) 19
- 関数リスト表示 (Purify、管理コード) 47
- コメント付きソース コード (PureCoverage) 64
- コメント付きソース コード (Quantify) 83
- コール グラフ (Purify、Java) 31, 38
- コール グラフ (Purify、管理コード) 46, 52
- コール グラフ (Quantify) 77, 84, 89
- コール グラフの差分ウィンドウ (Quantify) 84
- 差分の関数リスト (Quantify) 84
- データ ブラウザ (Purify、C/C++) 12–15, 16–19
- データ ブラウザ (Purify、Java) 31–33, 37
- データ ブラウザ (Purify、管理コード) 45–47, 50
- ナビゲータ (Purify、C/C++) 20
- ナビゲータ (Purify、Java) 31
- ナビゲータ (Purify、管理コード) 46
- ファイル表示 (Purify、カバレッジデータ) 19
- ラン サマリ (PureCoverage) 61
- ラン サマリ (Quantify) 88
- ウィンドウとタブモジュール表示  
(Purify、カバレッジデータ) 19

## え

- エラー (Purify、C/C++)
  - エラー データの保存 21
  - エラー発見時に中断 23
  - 修正 18
  - 分析 16
  - 「メッセージ」も参照 (Purify、C/C++)
- [エラー発見時に中断] ツール (Purify、C/C++) 23
- [エラー 表示] タブ、[データ ブラウザ] ウィンドウ (Purify、C/C++) 12
- 円グラフ、[関数詳細] ウィンドウ
  - Purify、Java 35
  - Purify、管理コード 49

## お

- オブジェクト参照
  - Java メモリ リーク 27
  - 管理コード メモリ リーク 42
- オブジェクト参照グラフ
  - Purify、Java 50
  - Purify、管理コード 36
- [オブジェクト詳細] ウィンドウ
  - Purify、Java 36
  - Purify、管理コード 50
- オブジェクト、分析
  - Purify、Java 35–37
  - Purify、管理コード 49–51
- [オブジェクト リスト表示] タブ
  - Purify、Java 37
  - Purify、管理コード 50

## か

- 開始
  - Purify、Java 30
- カスタマイズ
  - カバレッジ レベル (PureCoverage) 68
  - データ収集レベル (Quantify) 91
  - データ表示 (PureCoverage) 62
- カバレッジ データ (PureCoverage)
  - API 関数を使用した制御 70
  - 共有 66
  - コマンド ラインからの保存 71
  - 収集の制限 70

- フィルタ 63
- 複数ランのカバレッジ データのマージ 65
- ユーザー インターフェイスからの保存 65
- カバレッジ データ (Purify、C/C++)
  - 収集 10, 24
  - 保存 21
- カバレッジ データ収集の制限 (PureCoverage) 70
- カバレッジ データ ファイル (.cfy)
  - PureCoverage 66
  - Purify 21
- [カバレッジ ブラウザ] ウィンドウ (PureCoverage) 61
- カバレッジ モニター (Purify、C/C++)
  - /Coverage オプション 24
  - オン 10
  - 概説 9
  - カバレッジ データの使用法 18–20
  - カバレッジ データの保存 21
- カバレッジ レベル
  - カスタマイズ (PureCoverage) 68
  - デフォルトのレベルの設定 (PureCoverage) 68
- ガーベッジ コレクタ
  - Purify、Java 27, 31
  - Purify、管理コード 42, 45
- 関数
  - PureCoverage API 70
  - Purify API (C/C++) 25
  - Purify API (Java) 41
  - Purify API (管理コード) 55
  - Quantify API 88
- [関数詳細] ウィンドウ
  - Purify、Java 34
  - Purify、管理コード 48
  - Quantify 82
- 関数の下位関数、一覧表示 (Quantify) 82
- 関数の上位関数、一覧表示 (Quantify) 82
- 関数の処理時間 (Quantify) 80
- [関数リスト] ウィンドウ
  - 1 回のラン (Quantify) 80
  - 使用法 (PureCoverage) 62
  - データの並べ替え (Quantify) 80
  - ランの比較 (Quantify) 84
- 関数リストの負の値 (Quantify) 84
- 関数リスト表示
  - Purify、Java 33
  - Purify、カバレッジ データ 19
  - Purify、管理コード 47
- 関数レベルのインストールメンテーション (Purify) 21

- 関数レベルのカバレッジ
  - 概説 (PureCoverage) 61
  - 設定 (PureCoverage) 68
- 関数レベルのプロファイル (Quantify) 92
- 管理コード (PureCoverage)
  - コマンドラインからの実行 71
  - 対応言語 57
- 管理コード (Purify)
  - オブジェクトの分析 49–51
  - 管理コードを Purify する 44
  - メモリ使用状況グラフ 45
  - メモリ プロファイリング データのフィルタ 52
  - メモリ プロファイリング データの保存 51
  - メモリ リーク 42, 43
- 管理コード (Quantify)
  - 対応言語 73

## き

- 技術サポート、連絡先 5
- 起動
  - PureCoverage 60
  - Purify、C/C++ 10, 24, 54
  - Purify、管理コード 44
  - Quantify 75
- 基本手順
  - C/C++ コードを Purify する 9
  - Java コードを Purify する 29
  - 管理コードを Purify する 44
  - コード カバレッジの向上 59
  - プログラム パフォーマンスの向上 74
- キャッシュ ファイル
  - Purify、C/C++ 11
- 強調表示
  - 関連関数 (Quantify) 89
  - 関連メソッド (Purify、Java) 38
  - 関連メソッド (Purify、管理コード) 52
  - コール グラフでの緑色の強調表示 (Quantify) 84
  - パフォーマンスの向上 (Quantify) 84
- 行の色
  - コメント付きソース コード (PureCoverage) 64
  - コメント付きソース コード (Purify カバレッジデータ) 19
- 行の処理時間 (Quantify) 83
- 共有
  - データ ファイル (PureCoverage) 66
  - フィルタ (Purify、C/C++) 16

- 行レベルのインストゥルメンテーション (Purify、C/C++) 21
- 行レベルのカバレッジ (PureCoverage)
  - 概説 61
  - コメント付きソース コード 64
  - 設定 68
- 行レベルの測定 (Quantify) 92

## く

- [クイック フィルタ] コマンド (Purify、C/C++) 15
- グラフ
  - オブジェクト参照 (Purify、Java) 36
  - オブジェクト参照 (Purify、管理コード) 50
  - コール グラフ (Purify、Java) 31
  - コール グラフ (Purify、管理コード) 46
  - コール グラフ (Quantify) 77, 84, 89
  - メモリ使用状況グラフ (Purify、Java) 31
  - メモリ使用状況グラフ (Purify、管理コード) 45
- グループ、フィルタ (Purify、C/C++) 16

## こ

- コード
  - 編集 (Purify、C/C++) 18
  - 編集 (Purify、Java) 33
  - 編集 (Purify、管理コード) 47
- コマンド (Purify、C/C++)
  - データ ブラウザの埋め込み 23
- コマンド (Purify、Java)
  - サブツリー関連のコマンド 39
  - 展開/折りたたみ 39
  - フィルタ関連のコマンド 39
  - 元に戻す 39
- コマンド (Purify、管理コード)
  - サブツリー関連のコマンド 53
  - 展開/折りたたみ 53
  - フィルタ関連のコマンド 52
  - 元に戻す 53
- コマンド (Quantify)
  - サブツリー関連のコマンド 78, 90
  - 展開/折りたたみ 78, 90
  - フィルタ関連のコマンド 90
  - 元に戻す 90
- コマンドライン インターフェイス
  - Purify、C/C++ 24

- Purify、Java 40
- Purify、管理コード 54
- [コメント付きソース コード] ウィンドウ
- PureCoverage 64
- Purify (カバレッジ データ) 19
- Quantify 83
- コール グラフ (Purify、Java)
  - 概要 31
  - 関連メソッドの強調表示 38
  - サブツリー関連のコマンド 39
  - フィルタ関連のコマンド 39
- コール グラフ (Purify、管理コード)
  - 概要 46
  - 関連メソッドの強調表示 52
  - サブツリー関連のコマンド 52, 53
  - フィルタ関連のコマンド 52
- コール グラフ (Quantify)
  - 関連関数の強調表示 89
  - サブツリー関連のコマンド 78, 90
  - 初期設定の表示 77
  - 線の太さ 79
  - フィルタ関連のコマンド 90
  - ランの比較 84
- コール グラフでの緑色の強調表示 (Quantify) 84
- コール グラフのサブツリーの折りたたみ
  - Purify、Java 39
  - Purify、管理コード 53
  - Quantify 90
- コール グラフのサブツリーの削除
  - Purify、Java 39
  - Purify、管理コード 53
  - Quantify 80, 90
- コール グラフのサブツリーの展開
  - Purify、Java 39
  - Purify、管理コード 53
  - Quantify 78, 90
- コール グラフのサブツリーの非表示
  - Purify、Java 39
  - Purify、管理コード 53
  - Quantify 90
- コール グラフの差分ウィンドウ (Quantify) 84
- コール スタック (Purify、C/C++) 16, 17

## さ

- 最小インストゥルメンテーション
  - (Purify、C/C++) 21
- 再配置データ、インストゥルメンテーション

- Purify、C/C++ 10, 21, 44
- サブツリー (Purify、Java)
  - 削除 39
  - サブツリー関連のコマンドを元に戻す 39
  - 展開と折りたたみ 39
  - フォーカス 39
- サブツリー (Purify、管理コード)
  - 削除 53
  - サブツリー関連のコマンドを元に戻す 53
  - 展開と折りたたみ 53
  - フォーカス 53
- サブツリー (Quantify コール グラフ)
  - 削除 80, 90
  - サブツリー関連のコマンドを元に戻す 90
  - 展開と折りたたみ 90
  - フォーカス 90
- 差分の関数リスト (Quantify) 84

## し

- システム リソースとメモリ リーク
  - Purify、Java 28
  - Purify、管理コード 43
- 自動テストの Perl スクリプト
  - Quantify 92
- 自動テストのスクリプト
  - Quantify 92
- 自動テストのバッチ ファイル
  - Quantify 92
- 自動テスト用の Perl スクリプト
  - PureCoverage 71
  - Purify、C/C++ 24
  - Purify、Java 40
  - Purify、管理コード 54
- 自動テスト用のスクリプト
  - PureCoverage 71
  - Purify、C/C++ 24
  - Purify、Java 40
  - Purify、管理コード 54
- 自動テスト用のバッチ ファイル
  - PureCoverage 71
  - Purify、C/C++ 24
  - Purify、Java 40
  - Purify、管理コード 54
- 自動テスト用の makefile
  - PureCoverage 71
  - Purify、C/C++ 24
  - Purify、Java 40

Purify、管理コード 54  
Quantify 92  
自動マージ (PureCoverage) 65  
ジャストインタイム デバッグ (Purify、C/C++) 23  
終了時に使用中のハンドル (Purify、C/C++) 13  
終了メッセージ (Purify、C/C++) 13  
ショートカット メニュー  
    Purify、C/C++ 39  
    Purify、Java 13  
    Purify、管理コード 52  
処理時間の測定 (Quantify) 92

## す

スタック、コール (Purify、C/C++) 17  
ステータス バー、Quantify のウィンドウ 89  
スナップショット  
    カバレッジ データ (PureCoverage) 70  
    メモリ使用状況 (Purify、Java) 30  
    メモリ使用状況 (Purify、管理コード) 45  
スナップショットの差分  
    API と同等の結果 (Purify、Java) 41  
    API と同等の結果 (Purify、管理コード) 55  
    Purify、Java 31  
    Purify、管理コード 46  
スレッド ステータス、モニター (Quantify) 88

## せ

精密インストゥルメンテーション (Purify、C/C++) 21  
選択インストゥルメンテーション  
    PureCoverage 69  
    Quantify 87  
線の太さ、コール グラフ (Quantify) 79  
線の太さのスケーリング、Quantify コール グラフ 79

## そ

測定レベル (Quantify) 92  
ソース コード  
    表示 (PureCoverage) 64  
    表示 (Quantify) 83  
    編集 (Purify、C/C++) 18

編集 (Purify、Java) 33  
編集 (Purify、管理コード) 47  
ソース コードの編集  
    Purify、C/C++ 18  
    Purify、Java 33  
    Purify、管理コード 47

## た

対応言語とアプリケーション  
    PureCoverage 58  
    Purify、C/C++ 8  
    Quantify 73  
対応言語とコンポーネント  
    PureCoverage 58  
    Purify、C/C++ 8  
    Quantify 73  
対応プログラミング言語とコンポーネント  
    PureCoverage 58  
    Purify 7  
    Purify、C/C++ 8  
    Quantify 73

## つ

ツールチップ、コール グラフ  
    Purify、Java 32  
    Purify、管理コード 46

## て

テキスト ファイル (.txt)  
    PureCoverage 66, 71  
    Purify、C/C++ 21  
    Purify、Java 38  
    Purify、管理コード 51  
    Quantify 85  
テスト  
    自動テストでの PureCoverage の使用法 71  
    自動テストでの Purify の使用法 24, 26  
    自動テストでの Purify の使用法 (Java) 40  
    自動テストでの Quantify の使用法 92  
    ユニット テストでの PureCoverage の使用法 3  
    ユニット テストでの Purify の使用法 3  
    ユニット テストでの Quantify の使用法 3

## データ

「カバレッジ データ」、「エラー データ」、  
「メモリ プロファイリング データ」、  
「パフォーマンス データ」を参照

## データ収集の設定

PureCoverage 60

Quantify 75

## データの記録

制御 (Quantify) 88

デフォルト設定のレベルの変更 (Quantify) 91

プログラムによる制御 (Purify、Java) 41

プログラムによる制御 (Purify、管理コード) 55

## データの記録、制御

PureCoverage 70

Quantify 88

## データの並べ替え

PureCoverage 62

Quantify 80

## データの保存

コマンドラインからの保存 (PureCoverage) 71

コマンドラインからの保存  
(PureCoverage) 71, 92

コマンドラインからの保存

(Purify、C/C++) 25, 40, 55

コマンドラインからの保存 (Quantify) 92

ユーザー インターフェイスからの保存  
(PureCoverage) 65

ユーザー インターフェイスからの保存  
(Purify、C/C++) 21

ユーザー インターフェイスからの保存  
(Purify、Java) 38

ユーザー インターフェイスからの保存  
(Purify、管理コード) 51

## [データ ブラウザ] ウィンドウ (Purify)

エラー データ (C/C++) 12–20

オブジェクト リスト (Java) 37

オブジェクト リスト (管理コード) 50

カバレッジ データ (C/C++) 19

メモリ プロファイリング データ (Java) 31–33

メモリ プロファイリング データ  
(管理コード) 45–47

## [データ ブラウザの埋め込み] コマンド

(Purify、C/C++) 23

## デバッグ、ジャストインタイム (Purify、C/C++) 23

## デバッグ データ

インストゥルメンテーション (Purify、  
C/C++) 10, 21, 44

行レベルのカバレッジ (PureCoverage) 60

行レベルのプロファイル (Quantify) 75

## デフォルト設定のインストゥルメンテーション レベル、 設定

PureCoverage 68

Purify、C/C++ 21

Quantify 91

# と

## 統合

Microsoft Visual Basic (PureCoverage) 66

Microsoft Visual Basic (Quantify) 85

Microsoft Visual Studio 6 (PureCoverage) 66

Microsoft Visual Studio 6 (Purify、C/C++) 9–21

Microsoft Visual Studio 6 (Quantify) 85

Microsoft Visual Studio .NET (PureCoverage) 67

Microsoft Visual Studio .NET (Purify) 44

Microsoft Visual Studio .NET (Quantify) 85

Rational ClearQuest (PureCoverage) 66

Rational ClearQuest (Purify、C/C++) 25

Rational ClearQuest (Quantify) 85

Rational Robot (PureCoverage) 66

Rational Robot (Purify、C/C++) 25–26

Rational Robot (Quantify) 85

Rational Visual Test (PureCoverage) 66

Rational Visual Test (Purify、C/C++) 25–26

Rational Visual Test (Quantify) 85

# な

## ナビゲータ

PureCoverage 65

Purify、C/C++ 20

Purify、Java 31

Purify、管理コード 46

# は

## 配列の境界を越えた書き込みエラー

(Purify、C/C++) 16

## パフォーマンス データ (Quantify)

各行 83

関数のすべてのデータ 80

記録の制御 88

コマンドラインからの保存 92

単一の関数 82

パフォーマンスの向上の強調表示 84  
フィルタ 90  
ランの比較 84

## ひ

### 比較

スナップショット (Purify、Java) 31  
スナップショット (Purify、管理コード) 46  
プログラム ラン (Purify、C/C++) 20  
プログラム ラン (Purify、Java) 31  
プログラム ラン (Purify、管理コード) 46  
プログラム ラン (Quantify) 84

## ふ

### ファイル

.cfy (PureCoverage) 66  
.cfy (Purify、C/C++) 21  
.pcy (Purify、C/C++) 21  
.pft (Purify、C/C++) 16  
.pfy (Purify、C/C++) 21  
.pmy (Purify、Java) 38  
.pmy (Purify、管理コード) 51  
インストゥルメンテーション後の保存  
(キャッシュ) (Purify、C/C++) 11  
.txt (PureCoverage) 66  
.txt (Purify、C/C++) 21  
.txt (Purify、Java) 38  
.txt (Purify、管理コード) 51  
[ファイル表示] タブ (Purify、カバレッジデータ) 19

### フィルタ

.pft ファイルへの保存 (Purify、C/C++) 16  
概要 (Purify、C/C++) 38  
概要 (Purify、Java) 52  
概要 (Purify、管理コード) 14  
共有 (Purify、C/C++) 16  
フィルタ関連のコマンドを元に戻す  
(Purify、Java) 39  
フィルタ関連のコマンドを元に戻す  
(Purify、管理コード) 53  
フィルタ関連のコマンドを元に戻す  
(Quantify) 90  
フィルタ グループ (Purify、C/C++) 16  
フィルタ マネージャ (PureCoverage) 63  
フィルタ マネージャ (Purify、C/C++) 16

フィルタ マネージャ (Purify、Java) 40  
フィルタ マネージャ (Purify、管理コード) 53  
フィルタしたメッセージの表示 (Purify、C/C++) 16  
フィルタとサブツリー関連のコマンドを元に戻す  
Purify、Java 39  
Purify、管理コード 53  
Quantify 90

[フィルタの作成] コマンド (Purify、C/C++) 15  
複数ランのデータのマージ (PureCoverage) 65  
プログラム

Java プログラムの実行 (Purify) 29  
Microsoft Visual Studio 6 からの実行  
(PureCoverage) 67  
Microsoft Visual Studio 6 からの実行  
(Purify、C/C++) 10  
Microsoft Visual Studio 6 からの実行  
(Quantify) 85

Microsoft Visual Studio .NET からの実行  
(PureCoverage) 67

Microsoft Visual Studio .NET からの実行  
(Purify、管理コード) 44

Microsoft Visual Studio .NET からの実行  
(Quantify) 85

インストゥルメンテーション (PureCoverage) 60  
管理コード プログラムの実行 (Purify) 44

再実行 (Purify) 20

デバッガを使用して実行 (Purify、C/C++) 23

パフォーマンスのプロファイル (Quantify) 88  
プログラムの実行

PureCoverage 60

Purify、Java 30

Purify、管理コード 44

Purify 単独使用インターフェイスでの実行  
(Purify) 54

Purify 単独使用インターフェイスでの実行  
(Purify、C/C++) 24

Quantify 75

Visual Studio 6 からの実行 (PureCoverage) 67

Visual Studio 6 からの実行 (Purify、C/C++) 10

Visual Studio 6 からの実行 (Quantify) 85

Visual Studio .NET からの実行  
(PureCoverage) 67

Visual Studio .NET からの実行  
(Purify、管理コード) 44

Visual Studio .NET からの実行 (Quantify) 85

コマンド ラインからの実行 (Purify) 54

コマンド ラインからの実行 (Purify、C/C++) 24

コマンド ラインからの実行 (Purify、Java) 40

再実行 (Purify、C/C++) 20

プログラム パフォーマンスのプロファイル  
(Quantify) 88  
プログラム パフォーマンスのモニター (Quantify) 88  
プログラム ランのモニター (Quantify) 88  
プロシージャ レベルのカバレッジ (PureCoverage)  
「関数レベルのカバレッジ」を参照 (PureCoverage)

## め

メソッド、カテゴリごとの強調表示  
Purify、Java 38  
Purify、管理コード 52  
メソッド レベルのカバレッジ (PureCoverage)  
「関数レベルのカバレッジ」を参照 (PureCoverage)  
メッセージ (Purify、C/C++)  
展開 16  
フィルタ 14  
フィルタしたメッセージの再表示 16  
分析 16  
「エラー」も参照 (Purify、C/C++)  
メニュー、ショートカット (Purify、C/C++) 13  
メモリ使用状況グラフ  
Purify、Java 31  
Purify、管理コード 45  
メモリ プロファイリング データ (Purify)  
フィルタ (Java) 39  
フィルタ (管理コード) 52, 53  
保存 (Java) 38  
保存 (管理コード) 51  
メモリ リーク (Purify)  
Java メモリ リーク 27, 29  
PurifyNewLeaks API 関数 (C/C++) 25  
管理コード メモリ リーク 42, 43  
終了時に報告される C/C++ リーク 13

## も

モジュール  
インストウルメンテーションの制御  
(Purify、C/C++) 22  
インストウルメンテーション レベルの制御  
(Quantify) 91  
カバレッジからの除外 (PureCoverage) 68  
カバレッジ レベルの制御 (PureCoverage) 68  
モジュールごとのフィルタ (Purify、Java) 39

モジュールごとのフィルタ  
(Purify、管理コード) 52  
モジュールごとのフィルタ (Quantify) 90  
モジュールの除外 (PureCoverage) 68  
[モジュール表示] タブ  
(Purify カバレッジ データ) 19  
問題  
Java コード 27  
管理コード 42

## よ

呼び出しパス、コール グラフ (Quantify) 79

## ら

[ラン コントロール] ツールバー (Quantify) 88  
[ラン サマリ] ウィンドウ  
PureCoverage 61  
Quantify 88  
ラン、比較  
Purify、C/C++ 20  
Purify、Java 31  
Purify、管理コード 46  
Quantify 84

## り

リーク (Purify)  
Java 27  
管理コード 42  
「メモリ リーク (Purify)」も参照  
「メモリ リーク」も参照 (Purify)

## ん

.txt ファイル  
PureCoverage 66, 71  
Purify、C/C++ 21  
Purify、Java 38  
Purify、管理コード 51  
Quantify 85