Oracle® Solaris Studio 12.4：代码分析器用户指南

文件号码 E57231
2014 年 10 月

ORACLE®

版权所有 © 2014, Oracle 和/或其附属公司。保留所有权利。

本软件和相关文档是根据许可证协议提供的，该许可证协议中规定了关于使用和公开本软件和相关文档的各种限制，并受知识产权法的保护。除非在许可证协议中明确许可或适用法律明确授权，否则不得以任何形式、任何方式使用、拷贝、复制、翻译、广播、修改、授权、传播、分发、展示、执行、发布或显示本软件和相关文档的任何部分。除非法律要求实现互操作，否则严禁对本软件进行逆向工程设计、反汇编或反编译。

此文档所含信息可能随时被修改，恕不另行通知，我们不保证该信息没有错误。如果贵方发现任何问题，请书面通知我们。

如果将本软件或相关文档交付给美国政府，或者交付给以美国政府名义获得许可证的任何机构，必须符合以下规定：

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

本软件或硬件是为了在各种信息管理应用领域内的一般使用而开发的。它不应被应用于任何存在危险或潜在危险的应用领域，也不是为此而开发的，其中包括可能会产生人身伤害的应用领域。如果在危险应用领域内使用本软件或硬件，贵方应负责采取所有适当的防范措施，包括备份、冗余和其它确保安全使用本软件或硬件的措施。对于因在危险应用领域内使用本软件或硬件所造成的一切损失或损害，Oracle Corporation 及其附属公司概不负责。

Oracle 和 Java 是 Oracle 和/或其附属公司的注册商标。其他名称可能是各自所有者的商标。

Intel 和 Intel Xeon 是 Intel Corporation 的商标或注册商标。所有 SPARC 商标均是 SPARC International, Inc 的商标或注册商标，并应按照许可证的规定使用。AMD、Opteron、AMD 徽标以及 AMD Opteron 徽标是 Advanced Micro Devices 的商标或注册商标。UNIX 是 The Open Group 的注册商标。

本软件或硬件以及文档可能提供了访问第三方内容、产品和服务的方式或有关这些内容、产品和服务的信息。对于第三方内容、产品和服务，Oracle Corporation 及其附属公司明确表示不承担任何种类的担保，亦不对其承担任何责任。对于因访问或使用第三方内容、产品或服务所造成的任何损失、成本或损害，Oracle Corporation 及其附属公司概不负责。

目录

使用本文档 5

1 简介 7
- 代码分析器分析的数据 7
  - 静态代码检查 7
  - 动态内存访问检查 8
  - 代码覆盖检查 8
- 使用代码分析器的要求 8
- 代码分析器 GUI 9
- 代码分析器命令行界面 9
- 远程桌面分发 10
- 快速启动 10
  - ▼ 快速启动 10

2 收集数据和启动代码分析器 13
- 收集静态错误数据 13
- 收集动态内存访问数据 14
  - ▼ 如何从二进制文件收集动态内存访问数据： 14
- 收集代码覆盖数据 15
  - ▼ 如何从二进制文件收集代码覆盖数据： 15
- 使用代码分析器 GUI 16
- 使用代码分析器命令行工具 (`codean`) 17
  - `codean` 选项 17
  - `codean` 工作流示例 19

A 代码分析器分析的错误 21
- 代码覆盖问题 21
- 静态代码问题 21
  - 数组越界读 (ABR) 22
  - 数组越界写 (ABW) 22

- 双重释放内存 (DFM) 22
- 读取释放的内存 (FMR) 23
- 写入释放的内存 (FMW) 23
- 无限空循环 (INF) 23
- 内存泄漏 23
- 缺少函数返回值 (MFR) 23
- 缺少 malloc 返回值检查 (MRC) 24
- 泄漏指针检查器：Null 指针解除引用 (NUL) 24
- 返回释放的内存 (RFM) 25
- 读取未初始化的内存 (UMR) 25
- 未使用的返回值 (URV) 26
- 超出范围的局部变量使用 (VES) 26

动态内存访问错误 26
- 数组越界读 (ABR) 27
- 数组越界写 (ABW) 27
- 释放错误的内存块 (BFM) 27
- 错误的重新分配地址参数 (BRP) 28
- 损坏的保护块 (CGB) 28
- 双重释放内存 (DFM) 28
- 读取释放的内存 (FMR) 29
- 写入释放的内存 (FMW) 29
- 释放的重新分配参数 (FRP) 29
- 无效的内存读取 (IMR) 29
- 无效的内存写入 (IMW) 30
- 内存泄漏 30
- 重叠源和目标 (OLP) 30
- 部分初始化的读取 (PIR) 31
- 堆栈越界读 (SBR) 31
- 堆栈越界写 (SBW) 31
- 读取未分配的内存 (UAR) 31
- 写入未分配的内存 (UAW) 32
- 读取未初始化的内存 (UMR) 32

动态内存访问警告 32
- 分配零大小 (AZS) 33
- 内存泄漏 (MLK) 33
- 推测性内存读取 (SMR) 33

索引 35

使用本文档

- **概述** – 介绍如何使用代码分析器工具来分析和显示数据
- **目标读者** – 应用程序开发者、系统开发者、架构师、支持工程师
- **必备知识** – 编程经验、软件开发测试、生成和编译软件产品的经验

产品文档库

有关本产品的最新信息和已知问题均包含在文档库中，网址为：http://docs.oracle.com/cd/E37069_01。

获得 Oracle 支持

Oracle 客户可通过 My Oracle Support 获得电子支持。有关信息，请访问 http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info；如果您听力受损，请访问 http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs。

反馈

可以在 http://www.oracle.com/goto/docfeedback 上提供有关本文档的反馈。

♦ ♦ ♦ 第 1 章

简介

Oracle Solaris Studio 代码分析器是一套集成的工具，用于帮助 C 和 C++ 应用程序开发者针对 Oracle Solaris 开发出既安全又稳定的高质量软件。

本章包括有关下列内容的信息：

- "代码分析器分析的数据" [7]
- "使用代码分析器的要求" [8]
- "代码分析器 GUI" [9]
- "代码分析器命令行界面" [9]
- "远程桌面分发" [10]
- "快速启动" [10]

代码分析器分析的数据

代码分析器可以分析以下三种类型的数据：

- 编译期间检测到的静态代码错误
- 由内存错误搜索工具 `discover` 实用程序检测到的动态内存访问错误和警告
- 由代码覆盖工具 `uncover` 实用程序度量的代码覆盖数据

除了提供对各种类型的分析的访问以外，代码分析器还将静态代码检查与动态内存访问分析和代码覆盖分析集成，从而可以发现应用程序中其他单独使用的错误检测工具无法发现的许多重要错误。

代码分析器还可精确定位代码中的核心问题，修复这些问题后可能会消除其他问题。一个核心问题通常会伴有多个其他问题，例如，因为这些问题具有一个通用分配点或出现在同一函数中的同一数据地址上。

静态代码检查

静态代码检查可在编译期间检测代码中出现的常见编程错误。C 和 C++ 编译器的 `-xprevise=yes` 选项利用编译器的控制和数据流分析框架来分析应用程序是否存在潜在的编程和安全缺陷。

注 - 可以选择使用 `-xanalyze=code` 选项来收集静态代码错误，但此选项已停止使用。建议使用 `-xprevise=yes` 选项。

有关收集静态错误数据的信息，请参见“收集静态错误数据” [13]。

有关代码分析器分析的静态代码错误的列表，请参见“静态代码问题” [21]。

动态内存访问检查

通常，难以找到代码中与内存相关的错误。在运行程序之前使用 `discover` 对程序进行检测时，`discover` 会在程序执行期间动态捕捉并报告内存访问错误。例如，如果您的程序分配了一个数组但未将其初始化，然后尝试从数组中的某个位置执行读取操作，则该程序可能会出现异常行为。如果使用 Discover 对程序进行检测，然后运行此程序，`discover` 将捕捉该错误。

有关收集动态内存访问错误数据的信息，请参见“收集动态内存访问数据” [14]。

有关代码分析器分析的动态内存访问问题的列表，请参见“动态内存访问错误” [26]。

代码覆盖检查

代码覆盖可提供有关在测试中执行的和未执行的代码区域的信息，从而可以使您改进测试套件以测试更多代码。代码分析器可使用由 `uncover` 收集的数据来确定程序中哪些函数未覆盖，以及在添加覆盖相关函数的测试后要添加到应用程序总覆盖中的覆盖百分比。

有关收集代码覆盖数据的信息，请参见“收集代码覆盖数据” [15]。

使用代码分析器的要求

代码分析器可处理静态错误数据、动态内存访问错误数据，以及从使用 Oracle Solaris Studio 12.3 或 12.4 的 C 或 C++ 编译器编译的二进制文件收集的代码覆盖数据。

代码分析器在基于 SPARC 或 x86 的系统上运行，但要求其操作系统至少为 Solaris 10 10/08、Oracle Solaris 11、Oracle Enterprise Linux 5.x 或 Oracle Enterprise Linux 6.x。

代码分析器 GUI

使用编译器、Discover 或 Uncover 收集数据之后，通过发出 `code-analyzer` 命令，可以启动代码分析器 GUI 来显示并分析问题。

对于每个问题，代码分析器都会显示问题描述、发现该问题的源文件的路径名以及该文件中突出显示相关源代码行的代码片段。

使用代码分析器可以执行以下操作：

- 显示问题的更多详细信息。对于静态问题，详细信息包括“错误路径”。对于动态内存访问问题，详细信息包括“调用堆栈”，如果数据可用，还包括“分配堆栈”和“堆栈上的可用空间”。
- 打开发现问题的源文件。
- 从“错误路径”或堆栈中的函数调用移动到关联的源代码行。
- 查找函数在程序中的所有使用情况。
- 移到函数的声明
- 移动到被覆盖函数或覆盖函数的声明。
- 显示函数的调用图。
- 显示关于每个问题类型的更多信息，包括代码示例和可能的原因。
- 按分析类型、问题类型和源文件过滤显示的问题。
- 隐藏已查看的问题并关闭不需要关注的问题。

有关使用 GUI 的详细信息，请参见 GUI 中的联机帮助以及《Oracle Solaris Studio 12.4：代码分析器教程 》。

代码分析器命令行界面

代码分析器的命令行界面版本 `codean` 使用静态代码检查、Discover 和 Uncover，读取分析文件作为输入，并生成文本格式和 HTML 格式的输出。它还提供了一种机制，可以将数据存储在历史归档文件中，供以后对较新数据与历史数据进行比较。`codean` 可用于执行以下操作：

- 读入 API 格式的报告并将信息转换为文本和 HTML 格式。`codean` 将文本输出保存到 `.`*type*`.html` 文件，其中 *type* 可以是 `static`、`dynamic` 或 `coverage`。
- 对于 `.analyze/`*type*`/latest` 报告，可计算每个问题的检验和并将原始问题信息存储在 `.analyze/history//`*type* 文件中，其中 *type* 可以是 `static`、`dynamic` 或 `coverage`。
- 在最近的报告中仅显示新问题或已修复的问题，并将其与以前保存的报告进行比较。
- 指定要收集的数据类型：`dynamic`、`static`、`coverage` 或 `all`。
- 显示全路径名。

- 显示特定源文件中的问题。
- 显示源代码中的一定数量的行。
- 保存最近的报告。
- 覆盖具有相同标记名称的最近保存报告。
- 在报告中仅显示新问题或已修复的问题。
- 指定用于保存报告的目录。
- 过滤要显示的错误和警告类型。

有关更多信息，请参见 codean(1) 手册页。

远程桌面分发

可以为将在几乎任何操作系统上运行的代码分析器创建远程桌面分发，并在远程服务器上使用 Oracle Solaris Studio 编译器和工具。如果在安装过程中生成了远程桌面分发，并选中 "Export User Settings From Default Directory"（从缺省目录导出用户设置）选项，则代码分析器会将生成分发的服务器识别为远程主机，并访问 Oracle Solaris Studio 安装中的工具集合。缺省情况下此选项处于未选中状态。

要在远程操作系统上启动代码分析器，请运行相应的可执行文件：

```
./codeanalyzer/bin/codeanalyzer.exe
```

有关如何安装远程桌面分发的信息，请参见《Oracle Solaris Studio 12.4：安装指南》。

有关远程桌面分发的信息，请参见代码分析器 GUI 的联机帮助。

快速启动

以下示例使用样例 C 程序显示了收集有关代码的信息所需的步骤，以及如何使用代码分析器查看结果。

▼ 快速启动

1. **编译程序以收集静态数据。**

```
% cc -xprevise=yes *.c
```

注 - 以前，可以使用 -xanalyze=code 选项进行编译。此选项对 Oracle Solaris Studio 12.4 仍然有效，但已停止使用。

---

2. 使用调试信息重新编译程序。

```
% cc -g *.c
```

3. 使用 `discover` 检测程序，并运行程序以收集动态内存访问数据。

```
% cp a.out a.out.save
% discover -a a.out
% a.out
```

4. 使用 `uncover` 检测程序以收集代码覆盖数据。

```
% a.out
% cp a.out.save a.out
% a.out
% uncover a.out
```

5. 在收集信息之后，可以选择通过 GUI 或 `codean` 命令行工具访问代码分析器来显示所收集的数据。

   - 要通过 GUI 访问代码分析器，请使用以下命令：

     ```
     % code-analyzer a.out
     ```

   - 要通过命令行工具访问代码分析器，请使用以下命令：

     ```
     % codean a.out
     ```

♦ ♦ ♦ 第 2 章

收集数据和启动代码分析器

所收集的供代码分析器进行分析的数据存储在源代码文件所在目录的 *binary-name*.analyze 目录中。*binary-name*.analyze 目录是由编译器 discover 或 uncover 创建的。

本章包括有关下列主题的信息：

- "收集静态错误数据" [13]
- "收集动态内存访问数据" [14]
- "收集代码覆盖数据" [15]
- "使用代码分析器 GUI" [16]

收集静态错误数据

要收集 C 或 C++ 程序的静态错误数据，请在 Oracle Solaris Studio 12.3 或者 12.4 C 或 C++ 编译器中使用 -xprevise=yes 选项来编译该程序。以前使用 -xanalyze=code 选项，但此选项已停止使用，建议改用 -xprevise=yes 选项。在先前的 Oracle Solaris Studio 发行版中，-xprevise=yes 选项在编译器中不可用。使用此选项时，编译器会自动提取静态错误并将数据写入 *binary-name*.analyze 目录的 static 子目录中。

如果使用 -xprevise=yes 选项编译程序，然后在单独的步骤中链接此程序，那么您还需要在链接步骤中包括 -xanalyze=code 选项。

在 Linux 上，必须指定 -xannotate 选项和 -xprevise=yes 才能收集静态错误数据。例如：

```
% cc -xprevise=yes -xannotate -g t.c
```

请注意，编译器并不能检测到代码中的所有静态错误。

- 某些错误依赖于仅在运行时可用的数据。例如，假设给定以下代码，编译器不会检测 ABW（数组越界写）错误，因为它无法检测从某个文件读取的 ix 值是否位于 [0,9] 范围之外：

  ```
  void f(int fd, int array[10])
  {
    int ix;
  ```

```
    read(fd, &ix, sizeof(ix));
    array[ix] = 0;
  }
```

- 有些错误不明确，也可能不是实际错误。编译器不报告这些错误。
- 在此发行版中，编译器不检测某些复杂的错误。

收集静态错误数据之后，可以启动代码分析器的 GUI 或命令行工具 (codean) 来分析和显示数据，或者重新编译程序，以便可以收集动态内存访问或代码覆盖数据。

收集动态内存访问数据

收集 C 或 C++ 程序的动态内存访问数据的过程包含以下两个步骤：使用 discover 检测二进制文件，然后运行检测过的二进制文件。

要使用 discover 检测程序以收集数据供代码分析器使用，必须已使用 Oracle Solaris Studio 版本 12.3 或 12.4 的 C 或 C++ 编译器对程序进行了编译。使用 -g 选项进行编译可生成调试信息，从而使代码分析器可以显示动态内存访问错误和警告的源代码与行号信息。

如果在不进行优化的情况下编译程序，discover 将在源代码级别提供最完整的内存错误检测。如果编译时进行优化，将检测不到某些内存错误。

有关 Discover 能够检测或不能检测的特定类型二进制文件的信息，请参见《Oracle Solaris Studio 12.4：Discover 和 Uncover 用户指南 》中的“正确准备二进制文件”和《Oracle Solaris Studio 12.4：Discover 和 Uncover 用户指南 》中的“使用预装入或审计的二进制文件不兼容”。

注 - 您可以生成程序一次来同时用于 discover 和 uncover。但是，由于您不能检测已检测过的二进制文件，如果还打算使用 uncover 收集覆盖数据，请在使用 discover 进行检测之前先保存此二进制文件的副本以避免出现此问题。例如：

```
% cp a.out a.out.save
```

▼ 如何从二进制文件收集动态内存访问数据：

1. 使用 Discover 并结合 **-a** 选项检测二进制文件：

   % **discover -a** *binary_name*

注 - 必须使用 Oracle Solaris Studio 版本 12.3 或 12.4 中的 Discover 版本。-a 选项在早期的 discover 版本中不可用。

---

2. **运行检测过的二进制文件。**

   动态内存访问数据将被写入 *binary_name*`.analyze` 目录的 `dynamic` 子目录中。

   注 - 有关在使用 `discover` 检测二进制文件时可以指定的其他检测选项，请参见《Oracle Solaris Studio 12.4：Discover 和 Uncover 用户指南 》中的"检测选项"或 `discover` 手册页。

   ---

3. **（可选） 启动代码分析器的 GUI 或命令行工具 (`codean`) 以分析和显示数据，以及以前可能收集的任何静态代码数据。或者，可以使用未经检测的二进制文件副本来收集代码覆盖数据。**

收集代码覆盖数据

收集有关 C 或 C++ 程序的代码覆盖数据的过程分为三步：

1. 使用 `uncover` 检测二进制文件
2. 运行检测过的二进制文件
3. 再次运行 `uncover` 以生成覆盖报告供代码分析器使用。

可以在检测二进制文件后多次运行该检测过的二进制文件，并累积各次运行所获取的数据，然后生成覆盖报告。

▼ 如何从二进制文件收集代码覆盖数据：

开始之前 要使用 `uncover` 检测程序以收集数据供代码分析器使用，必须已使用 Oracle Solaris Studio 版本 12.3 或 12.4 的 C 或 C++ 编译器对程序进行了编译。使用 `-g` 选项进行编译可生成调试信息，从而使代码分析器可以使用源代码级别覆盖信息。

注 - 如果在编译程序以便使用 `discover` 进行检测时保存了二进制文件副本，可以将该副本重命名为原始二进制文件名，以供在使用 `uncover` 进行检测时使用。例如：

```
cp a.out.save a.out
```

---

1. **使用 Uncover 检测二进制文件：**

   % **uncover** *binary-name*

2. **运行检测过的二进制文件一次或多次。**

   代码覆盖数据将被写入 *binary-name*`.uc` 目录。

3. 结合使用 Uncover 与 -a 选项基于累积的数据生成代码覆盖报告：

   % **uncover -a** *binary-name*.**uc**

   覆盖报告将被写入 *binary-name*.`analyze` 目录的 coverage 子目录中。

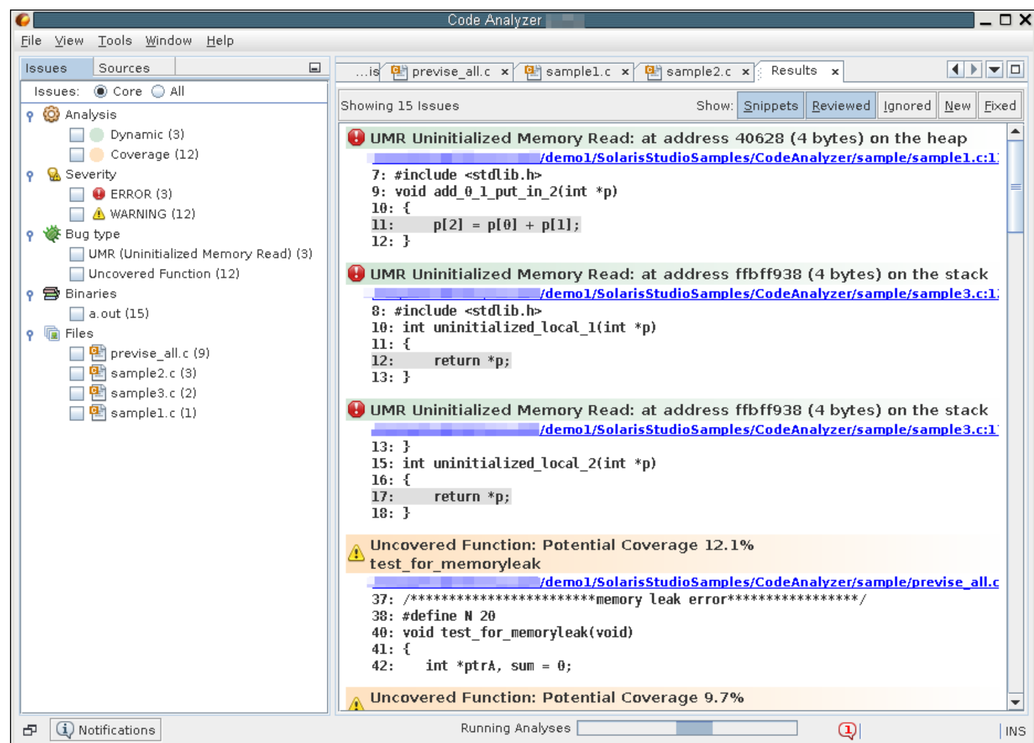
   注 - 必须使用 Oracle Solaris Studio 版本 12.3 或 12.4 中的 `uncover` 版本。-a 选项在早期的 `uncover` 版本中不可用。

使用代码分析器 GUI

可以使用代码分析器 GUI 分析最多三种类型的数据。要启动 GUI，请键入 `code-analyzer` 命令以及要为其分析所收集错误数据的二进制文件的路径：

% **code-analyzer** *binary-name*

此时将打开代码分析器 GUI 并显示 *binary-name*.`analyze` 目录中的数据，如下图所示。



运行代码分析器 GUI 时，可以通过选择 "Open"（打开）-> "File"（文件）并导航到另一个二进制文件来切换显示为该文件收集的数据。

GUI 中的联机帮助介绍了如何使用所有功能来过滤显示的结果、显示或隐藏问题以及显示有关特定问题的更多信息。《Oracle Solaris Studio 12.4：代码分析器教程 》通过使用一个样例程序来指导您完成完整的数据收集和分析方案。

使用代码分析器命令行工具 (codean)

也可以使用代码分析器命令行工具 `codean` 分析最多三种类型的数据。要启动 `codean`，请键入 `codean` 命令、任何选项以及可执行文件或目录的路径。

`codean` *options executable-path*|*directory*

`codean` 工具在屏幕上显示文本输出。也可以在可执行文件所在位置中的 `.`*type*`.html` 文件中查看结果。本节介绍命令选项。

codean 选项

以下各节介绍可用于 `codean` 的不同选项。

数据类型选项

以下选项确定要收集的数据类型。

`-s` 处理和显示静态数据。

`-d` 处理和显示动态数据。

`-c` 处理和显示覆盖数据。

可以指定多个选项，也可以不指定任何选项。如果不指定任何选项，则缺省为处理所有可能的选项，具体取决于 `.analyze/`*type*`/latest` 文件是否存在；其中 *type* 可以是 `static`、`dynamic` 或 `coverage`。

显示选项

以下选项确定包含结果的文本输出的内容。

`--fullpath`	显示文件的完整路径名称。
`-f` *source-file*	仅显示指定源文件中的问题。
`-n` *number*	显示指定行数的源代码。

过滤选项

以下选项确定在结果中报告的错误和警告的类型。

错误或警告类型可以是以下类型之一：

- 三个字母的错误代码或三个字母的警告代码。有关可能的错误和警告的列表，请参见附录 A, 代码分析器分析的错误。
- `MLK` 或 `mlk`，表示内存泄漏。
- `ALL` 或 `all`，表示所有警告或错误。

如果未指定错误或警告，则缺省值为 `all`。

过滤选项包括：

`--showerrors` *error-type*	仅显示指定错误类型的错误。
`--showwarnings` *warning-type*	仅显示指定警告类型的警告。
`--hideerrors` *error-type*	不显示指定错误类型的错误。
`--hidewarnings` *warning-type*	不显示指定警告类型的警告。

保存结果选项

可以将最新的结果保存在文件中，放在具有特定标记名称的特定目录中。

`--save`	保存最近的报告。
`--tag` *tag-name*	与 `--save` 配合使用时，用标记名称 *tag-name* 为保存的副本命名。如果某个已保存副本具有相同的标记名称，则 `codean` 将发出警告消息，然后退出而不覆盖文件。如果未指定标记名称，则 `codean` 将检

	查可执行文件的最近报告的最近修改时间，并使用该时间戳作为标记名称。
`-t`	覆盖具有相同标记名称的已保存报告。
`-D` *directory*	将报告保存到目录 *directory*。

比较结果选项

通过以下选项可以将结果与以前生成的报告进行比较。

`--whatisnew`	仅显示新问题。此选项不能与 `--whatisfixed` 一起使用。
`--whatisfixed`	仅显示已修复的问题。此选项不能与 `--whatisnew` 一起使用。
`--tag` *tag-name*	与 `--whatisnew` 或 `--whatisfixed` 配合使用时，将使用标记名称为 *tag-name* 的报告的历史副本与新生成的报告进行比较。如果未指定标记名称，则将最近的报告与最近保存的副本进行比较。
`--ref` *file*|*directory*	必须与 `--whatisnew` 或 `--whatisfixed` 配合使用，并且必须后跟一个路径名。此选项指定要与新报告进行比较的文件或目录。

codean 工作流示例

本节提供一个监视错误修复效果的示例。

1. 在修复之前编译目标源。

```
% cc -g *.c
```

2. 使用 Discover 检测二进制文件，并确保其生成 Analytics 输出。

```
% discover -a a.out
```

3. 运行检测过的二进制文件。
4. 使用 `codean` 存储 Analytics 输出。将在 `a.out.analyze/history/before_bugfix` 中创建一个历史归档文件，还会在此目录中创建一个名为 `dynamic` 的历史文件。

```
% codean --save --tag before_bugfix -d a.out
```

5. 修复错误。
6. 再次编译目标源代码。

```
% cc -g *.c
```

7. 再次使用 `discover` 检测二进制文件。

```
% discover -a a.out
```

8. 运行检测过的二进制文件。

```
% a.out
```

9. 显示比较结果，并确保错误所导致的无效内存访问已被修复。

```
% codean --whatisfixed --tag before_bugfix -d a.out
```

   这将生成一个新的 Analytics 输出文件（位于 a.out.analyze/dynamic/fixed_before_bugfix，其中仅包含已修复的动态问题。可以使用 codean 或代码分析器 GUI 来查看这些已修复的问题。

10. （可选）运行 codean 以确保未引入任何新错误。

```
% codean --whatisnew --tag before_bugfix -d a.out
```

   此命令生成一个新的 Analytics 文件（位于 a.out.analyze/dynamic/new_before_bugfix），其中仅包含新的动态问题。

♦ ♦ ♦ 附 录 A

代码分析器分析的错误

编译器、discover 和 uncover 可在您的代码中查找静态代码问题、动态内存访问问题以及覆盖问题。本附录介绍了由这些工具发现并由代码分析器分析的特定问题类型。

- “代码覆盖问题” [21]
- “静态代码问题” [21]
- “动态内存访问错误” [26]
- “动态内存访问警告” [32]

代码覆盖问题

代码覆盖检查可确定哪些函数未被覆盖。在结果中，发现的代码覆盖问题会标记为 "Uncovered Function"（未覆盖的函数），并且带有潜在覆盖百分比，此百分比是指在添加覆盖相关函数的测试后要添加到应用程序总覆盖中的覆盖百分比。

可能的原因：任何测试都无法执行您的函数，或者您可能忘记了删除死代码或旧代码。

静态代码问题

静态代码检查可查找以下类型的错误：

- ABR：数组越界读
- ABW：数组越界写
- DFM：双重释放内存
- ECV：显式强制类型转换违规
- FMR：读取释放的内存
- FMW：写入释放的内存
- INF：无限空循环
- MLK：内存泄漏
- MFR：缺少函数返回值

- MRC：缺少 malloc 返回值检查
- NFR：返回未初始化的函数
- NUL：NULL 指针解除引用，泄漏指针检查
- RFM：返回释放的内存
- UMR：读取未初始化的内存。未初始化的内存读取位操作
- URV：未使用的返回值
- VES：超出范围的局部变量使用

本节介绍可能的错误原因以及发生错误的代码示例。

数组越界读 (ABR)

可能的原因：试图读取超出数组边界的内存。

示例：

```
int a[5];
. . .
printf("a[5] = %d\n",a[5]);  // Reading memory beyond array bounds
```

数组越界写 (ABW)

可能的原因：试图写入超出数组边界的内存。

示例：

```
int a [5];
 . . .
 a[5] = 5; // Writing to memory beyond array bounds
```

双重释放内存 (DFM)

可能的原因：多次使用同一指针调用 `free()`（）。在 C++ 中，多次对同一指针使用 `delete` 操作符。

示例：

```
int *p = (int*) malloc(sizeof(int));
 free(p);
 . . .      // p was not signed a new value between the free statements
 free(p); // Double freeing memory
```

读取释放的内存 (FMR)

示例：

```
int *p = (int*) malloc(sizeof(int));
free(p);
. . .  // Nothing assigned to p in between
printf("p = 0x%h\n",p); // Reading from freed memory
```

写入释放的内存 (FMW)

示例：

```
int *p = (int*) malloc(sizeof(int));
 free(p);
 . . .        // Nothing assigned to p in between
 *p = 1; // Writing to freed memory
```

无限空循环 (INF)

示例：

```
int x=0;
int i=0;
while (i200) {
  x++; } // Infinite loop
```

内存泄漏

可能的原因：分配了内存，但是在退出或终止函数之前未释放。

示例：

```
int foo()
{
 int *p = (int*) malloc(sizeof(int));
 if (x) {
  p = (int *) malloc(5*sizeof(int));  // will cause a leak of the 1st malloc
  }
}                                      // The 2nd malloc leaked here
```

缺少函数返回值 (MFR)

可能的原因：沿某些路径退出时缺少返回值。

示例：

```
#include <stdio.h>
int foo (int a, int b)
{
  if (a)
    {
       return b;
    }
}            // If foo returns here, the return is uninitialized
int main ( )
{
    printf("%d\n", foo(0,30));
}
```

缺少 malloc 返回值检查 (MRC)

可能的原因：访问 C 中 `malloc` 或 C++ 中新运算符的返回值时未执行 `null` 检查。

示例：

```
#include <stdlib.h>
int main()
{
 int *p3 = (int*) malloc(sizeof(int)); // Missing null-pointer check after malloc.
 *p3 = 0;
}
```

泄漏指针检查器：Null 指针解除引用 (NUL)

可能的原因：访问可能为 `null` 的指针，或在指针从不为 null 的情况下针对 `null` 进行冗余检查。

示例：

```
#include <stdio.h>
#include <stdlib.h>
int gp, ctl;
int main()
{
 int *p = gp;
 if (ctl)
  p = 0;
 printf ("%c\n", *p); // May be null pointer dereference
 if (!p)
  *p = 0; // Surely null pointer dereference

int *p2 = gp;
```

```
*p2 = 0; // Access before checking against NULL.
assert (p2!=0);

int *p3 = gp;
if (p3) {
  printf ("p3 is not zero.\n");
}
*p3 = 0; // Access is not protected by previous check against NULL.
}
```

返回释放的内存 (RFM)

示例：

```
#include <stdlib.h>
int *foo ()
{
 int *p = (int*) malloc(sizeof(int));
 free(p);
 return p; // Return freed memory is dangerous
}
int main()
{
 int *p = foo();
 *p = 0;
}
```

读取未初始化的内存 (UMR)

可能的原因：读取尚未初始化的局部数据或堆数据。

示例：

```
#include <stdio.h>
#include <stdlib.h>
 struct ttt {
    int a: 1;
    int b: 1;
 };

 int main()
 {
   int *p = (int*) malloc(sizeof(int));
   printf("*p = %d\n",*p); // Accessing uninitialized data

   struct ttt t;
   extern void foo (struct ttt *);

   t.a = 1;
```

```
    foo (&t); // Access uninitialized bitfield data "t.b"
  }
```

未使用的返回值 (URV)

可能的原因：读取尚未初始化的局部数据或堆数据。

示例：

```
 int foo();
int main()
{
   foo(); // Return value is not used.
}
```

超出范围的局部变量使用 (VES)

可能的原因：读取尚未初始化的局部数据或堆数据。

示例：

```
 int main()
{
   int *p = (int *)0;
   void bar (int *);
   {
     int a[10];
     p = a;
   } // local variable 'a' leaked out
   bar(p);
 }
```

动态内存访问错误

动态内存访问检查查找以下类型的错误：

- ABR：数组越界读
- ABW：数组越界写
- BFM：释放错误的内存块
- BRP：错误的重新分配地址参数
- CGB：损坏的保护块
- DFM：双重释放内存
- FMR：读取释放的内存

- FMW：写入释放的内存
- FRP：释放的重新分配参数
- IMR：无效的内存读取
- IMW：无效的内存写入
- MLK：内存泄漏
- OLP：重叠源和目标
- PIR：部分初始化的读取
- SBR：堆栈越界读
- SBW：堆栈越界写
- UAR：读取未分配的内存
- UAW：写入未分配的内存
- UMR：读取未初始化的内存

本节介绍可能的错误原因以及发生错误的代码示例。

数组越界读 (ABR)

可能的原因：试图读取超出数组边界的内存。

示例：

```
int a[5];
. . .
printf("a[5] = %d\n",a[5]);  // Reading memory beyond array bounds
```

数组越界写 (ABW)

可能的原因：试图写入超出数组边界的内存。

示例：

```
int a [5];
 . . .
 a[5] = 5; // Writing to memory beyond array bounds
```

释放错误的内存块 (BFM)

可能的原因：向 `free()` () 或 `realloc()` () 传递了一个非堆数据指针。

示例：

```
#include <stdlib.h>
int main()
{
 int *p = (int*) malloc(sizeof(int));
 free(p+1); // Freeing wrong memory block
}
```

错误的重新分配地址参数 (BRP)

示例：

```
#include <stdlib.h>
int main()
{
 int *p = (int*) realloc(0,sizeof(int));
 int *q = (int*) realloc(p+20,sizeof(int[2])); // Bad address parameter for realloc
}
```

损坏的保护块 (CGB)

可能的原因：写入超出了动态分配的数组末尾，或者在“红色区域”中。

示例：

```
#include <stdio.h>
#include <stdlib.h>

int main() {
 int *p = (int *) malloc(sizeof(int)*4);
 *(p+5) = 10; //  Corrupted array guard block detected (only when the code is not
annotated)
 free(p);

 return 0;
}
```

双重释放内存 (DFM)

可能的原因：多次使用同一指针调用 `free()` （）。在 C++ 中，多次对同一指针使用 `delete` 操作符。

示例：

```
int *p = (int*) malloc(sizeof(int));
 free(p);
 . . .      // p was not assigned a new value between the free statements
```

```
free(p); // Double freeing memory
```

读取释放的内存 (FMR)

示例：

```
int *p = (int*) malloc(sizeof(int));
 free(p);
 . . .  // Nothing assigned to p in between
 printf("p = 0x%h\n",p); // Reading from freed memory
```

写入释放的内存 (FMW)

示例：

```
int *p = (int*) malloc(sizeof(int));
 free(p);
 . . .        // Nothing assigned to p in between
 *p = 1; // Writing to freed memory
```

释放的重新分配参数 (FRP)

示例：

```
#include <stdlib.h>

int main() {
  int *p = (int *) malloc(sizeof(int));
  free(0);
  int *q = (int*) realloc(p,sizeof(it[2])); //Freed pointer passed to realloc
}
```

无效的内存读取 (IMR)

可能的原因：分别从那些没有半字对齐、单字对齐或双字对齐的地址中读取 2 个、4 个或 8 个字节。

示例：

```
#include <stdlib.h>
int main()
{
  int *p = 0;
  int i = *p;   // Read from invalid memory address
```

```
}
```

无效的内存写入 (IMW)

可能的原因：分别从那些没有半字对齐、单字对齐或双字对齐的地址中写入 2 个、4 个或 8 个字节。向文本地址写入、向只读数据区 (.rodata) 写入或向已由 mmap 设置为只读的页面写入。

示例：

```
  int main()
{
  int *p = 0;
  *p = 1;  // Write to invalid memory address
}
```

内存泄漏

可能的原因：分配了内存，但是在退出或终止函数之前未释放。

示例：

```
    int foo()
{
  int *p = (int*) malloc(sizeof(int));
  if (x) {
   p = (int *) malloc(5*sizeof(int));  // will cause a leak of the 1st malloc
  }
}                                      // The 2nd malloc leaked here
```

重叠源和目标 (OLP)

可能的原因：指定的源、目标或长度不正确。如果源和目标重叠，则程序的行为是不确定的。

示例：

```
#include <stlib.h>
#include <string.h>
int main() {
  char *s=(char *) malloc(15);
  memset(s, 'x', 15);
  memcpy(s, s+5, 10);
  return 0;
}
```

部分初始化的读取 (PIR)

示例：

```
 #include <stdio.h>
#include <stdlib.h>
 int main()
{
  int *p = (int*) malloc(sizeof(int));
  *((char*)p) = 'c';
  printf("*(p = %d\n",*(p+1)); // Accessing partially initialized data
}
```

堆栈越界读 (SBR)

可能的原因：读取本地数组时越过了起始或结束边界。

示例：

```
 #include <stdio.h>

int main() {
  int a[2] = {0, 1};
  printf("a[-10]=%d\n",a[-10]); //  Read is beyond stack frame bounds

  return 0;
}
```

堆栈越界写 (SBW)

可能的原因：写入本地数组时越过了起始或结束边界。

示例：

```
 #include <stdio.h>

 int main() {
  int a[2] = {0, 1};
  a[-10] = 2; //  Write is beyond stack frame bounds

  return 0;
}
```

读取未分配的内存 (UAR)

可能的原因：溢出堆块边界或访问已释放堆块的迷失指针。

示例：

```
 #include <stdio.h>
 #include <stdlib>
int main()
{
  int *p = (int*) malloc(sizeof(int));
  printf("*(p+1) = %d\n",*(p+1)); // Reading from unallocated memory
}
```

写入未分配的内存 (UAW)

可能的原因：溢出堆块边界或访问已释放堆块的迷失指针。

示例：

```
 #include <stdio.h>
 #include <stdlib>
int main()
{
 int *p = (int*) malloc(sizeof(int));
  *(p+1) = 1; // Writing to unallocated memory
}
```

读取未初始化的内存 (UMR)

可能的原因：读取尚未初始化的局部数据或堆数据。

示例：

```
 #include <stdio.h>
 #include <stdlib>
int main()
{
 int *p = (int*) malloc(sizeof(int));
 printf("*p = %d\n",*p); // Accessing uninitialized data
}
```

动态内存访问警告

动态内存访问检查可查找以下类型的警告：

- AZS：分配零大小
- 内存泄漏
- SMR：推测性未初始化内存读取

本节介绍可能的警告原因以及可能发生警告的代码示例。

分配零大小 (AZS)

示例：

```
#include <stdlib>
int main()
{
  int *p = malloc(); // Allocating zero size memory block
}
```

内存泄漏 (MLK)

可能的原因：分配了内存，但是在退出或终止函数之前未释放。

示例：

```
int foo()
{
 int *p = (int*) malloc(sizeof(int));
 if (x) {
  p = (int *) malloc(5*sizeof(int));  // will cause a leak of the 1st malloc
 }
}                                     // The 2nd malloc leaked here
```

推测性内存读取 (SMR)

示例：

```
int i;
if (foo(&i) != 0)  /* foo returns nonzero if it has initialized i */
printf("5d\n", i);
```

编译器可能会针对上面的源代码生成下面的等效代码：

```
   int i;
   int t1, t2'
   t1 = foo(&i);
   t2 = i; /* value in i is loaded. So even if t1 is 0, we have uninitialized read due to
speculative load */
   if (t1 != 0)
   printf("%d\n", t2);
```

索引

B
binary_name.analyze 目录
  dynamic 子目录，15
binary-name.analyze 目录，13，16
  coverage 子目录，16
  static 子目录，13

C
codean
  功能，9
codean 命令，9

D
代码分析器
  使用要求，8
代码分析器 GUI
  功能，9
  快速启动，10
  启动，16
代码分析器命令行界面
  功能，9
代码覆盖检查，8
代码覆盖问题，21
动态内存访问检查，8
动态内存访问问题
  错误，26
  警告，32

G
-g 编译器选项，14，15

H
核心问题，7

J
检测程序
  使用 Discover，14
  使用 discover，14
  使用 Uncover，15，15
静态代码检查，7
静态代码问题，21

S
收集数据
  代码覆盖，15
  动态内存访问错误，14
  静态错误，13
    限制，13
  binary-name.analyze 目录，13

X
选项，17
  保存结果，18
  比较结果，19
  过滤，18
  数据类型，17
  显示，17
-xanalyze=code 编译器选项，7，13，13
  Linux，13
-xprevise=yes 编译器选项，7，13，13
  Linux，13

Y

要求
  使用 Discover 检测程序，14
  使用 `uncover` 检测程序，15
  使用代码分析器，8
优化, 对内存错误的影响，14