Oracle® Solaris Studio 12.4：Discover 和 Uncover 用户指南

文件号码 E57227
2015 年 12 月

ORACLE

文件号码 E57227

文档可访问性

有关 Oracle 对可访问性的承诺，请访问 Oracle Accessibility Program 网站 http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc。

获得 Oracle 支持

购买了支持服务的 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。

目录

使用本文档

- **概述** – 介绍如何使用内存错误搜索工具 (discover) 以二进制形式查找内存有关的错误，以及使用代码覆盖工具 (uncover) 测量应用程序的代码覆盖。
- **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

有关该产品及相关产品的文档和资源，可从以下网址获得：http://docs.oracle.com/cd/E37069_01。

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

1 简介

《*Oracle Solaris Studio 12.4 Discover* 和 *Uncover* 用户指南》介绍如何使用以下工具：

内存错误搜索工具 (discover)

内存错误搜索工具 (discover) 软件是用于检测内存访问错误的高级开发工具。discover 实用程序适用于使用 Sun Studio 12 Update 1、Oracle Solaris Studio 12.2、Oracle Solaris Studio 12.3 或 Oracle Solaris Studio 12.4 编译器编译的二进制文件。它可以在基于 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。

程序中与内存相关的错误极难发现。通过 discover 实用程序，您可以通过定位源代码中出现问题的确切位置来轻松地找到此类错误。例如，如果您的程序分配了一个数组，但未将其初始化，然后尝试从一个数组位置执行读取操作，则该程序可能会出现异常行为。当您以正常方式运行程序时，discover 实用程序可以捕捉到此问题。

discover 可以检测到的其他错误包括：

- 对未分配的内存执行读写
- 访问超出分配数组边界的内存
- 不正确地使用释放的内存
- 释放错误的内存块
- 内存泄漏

由于 discover 是在程序执行期间动态捕捉并报告内存访问错误，因此，如果运行时用户代码的某个部分未执行，则不会报告该部分的错误。

discover 实用程序的用法很简单。编译器所准备的任何二进制文件（甚至是完全优化的二进制文件）均可使用单个命令进行检测，然后以正常方式运行。运行期间，discover 会生成内存异常报告，您可以在 Web 浏览器中以文本文件或 HTML 格式查看该报告。

代码覆盖工具 (uncover)

uncover 实用程序是一个简单易用的命令行工具，用于度量应用程序的代码覆盖。代码覆盖是软件测试的重要组成部分。该工具提供了测试时执行的具体代码区域的相关信息，使您可以改进测试套件以测试更多代码。uncover 可以报告函数、语句、基本块或指令级别的覆盖信息。

uncover 实用程序提供了一个称为“未覆盖”的独特功能，可帮助您快速找到未测试的主要功能区域。与其他类型的检测相比，uncover 代码覆盖的其他优势包括：

- 相对于未检测的代码而言，性能只有些许的下降。
- 由于 uncover 会处理二进制文件，因此，它可以处理任何优化的二进制文件。
- 只需通过检测随附的二进制文件即可完成度量。要进行覆盖测试，无需以不同的方式生成应用程序。
- uncover 实用程序提供了一套检测二进制文件、运行测试和显示结果的简单过程。
- uncover 实用程序是多线程安全的，并且是多进程安全的。

♦ ♦ ♦ 第 2 章

2 内存错误搜索工具 (discover)

内存错误搜索工具 (discover) 软件是用于检测内存访问错误的高级开发工具。本章包括有关下列内容的信息：

使用 discover 的要求

本节介绍使用 discover 并取得最佳结果的要求，包含以下主题：

正确准备二进制文件

discover 实用程序适用于使用 Sun Studio 12 Update 1、Oracle Solaris Studio 12.2、Oracle Solaris Studio 12.3 或 Oracle Solaris Studio 12.4 编译器编译的二进制文件。它可以在基于 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。

如果不满足这些要求，discover 实用程序会发出错误并且不检测二进制文件。不过，您可以检测不满足这些要求的二进制文件并使用 -l 选项检测有限数量的错误。请参见“检测选项” [17]。

编译后的二进制文件包含称为注释的信息，以帮助 discover 正确对其进行检测。添加这些少量信息不会影响二进制文件的性能或运行时内存使用情况。

使用 -g 选项可在编译二进制文件时生成调试信息，因此 discover 可在报告错误和警告时显示源代码和行号信息并生成更准确的结果。如果在编译二进制文件时未使用 -g 选项，discover 将仅显示相应计算机级别指令的程序计数器。此外，使用 -g 选项进行编译还可帮助 discover 生成更准确的报告。尽管 discover 可与许多优化的二进制文件一起使用，但仍建议使用 -g。有关更多信息，请参见“解释 discover 错误消息” [43]。

要获取最佳结果，编译二进制文件时应使用 -g 选项且不使用任何优化选项。优化代码可能会由于优化而不同于源代码，例如，对不同的变量使用相同的内存位置以及生成推测性代码。编译时使用高级优化选项可能会导致 discover 不正确地报告错误或者不报告错误。

注 - discover 支持重新定义标准内存分配函数（malloc（）、calloc（）、memalign（）、valloc（） 和 free（））的二进制文件。

有关更多信息，请参见“使用 discover 时的限制” [46]

使用预装入或审计的二进制文件不兼容

由于 discover 使用了运行时链接程序的某些特殊功能，因此您无法将其用于使用预装入或审计的二进制文件。

如果程序要求设置 LD_PRELOAD 环境变量，则该变量可能无法与 discover 一起正常运行，因为 discover 需要在某些系统函数上插入，但如果函数已预装入，将无法执行插入。

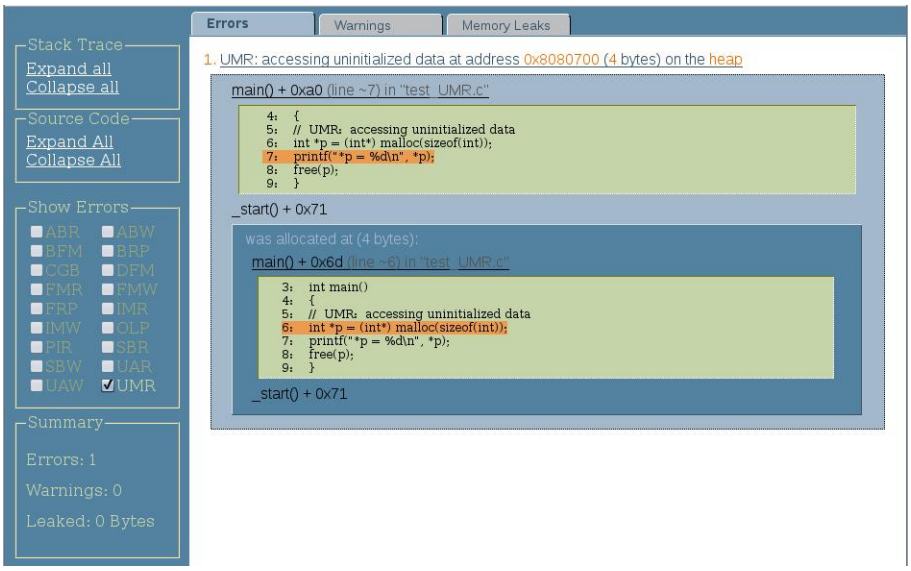
类似地，如果程序使用运行时审计，则由于二进制文件已通过 -p 选项或 -P 选项被链接，或者二进制文件要求设置 LD_AUDIT 环境变量，因此该审计将与 discover 使用的审计相冲突。如果在链接二进制文件时使用了审计，则 discover 会在检测时失败。如果在运行时设置了 LD_AUDIT 环境变量，结果将无法确定。

简单程序示例

以下示例说明了如下过程：准备程序，使用 discover 对其检测，然后运行该程序并生成有关检测到的内存访问错误的报告。此示例使用了一个访问未初始化数据的简单程序。

```
% cat test_UMR.c
#include <stdio.h>
#include <stdlib.h>
int main()
{
//  UMR:  accessing uninitialized data
int *p = (int*) malloc(sizeof(int));
printf("*p = %d\n", *p);
free(p);
}

% cc -g -02 test_UMR.c
% a.out
*p = 131464
% discover a.out
% a.out
```

discover 输出指示在何处使用了未初始化的内存以及在何处对其进行了分配，同时提供结果摘要，如下图中所示。



检测准备好的二进制文件

准备好目标二进制文件后，下一步就是对其进行检测。检测会在关键位置添加代码，以便 discover 可以在二进制文件运行时跟踪内存操作。

注 - 对于 SPARC V8 体系结构上的 32 位二进制文件，discover 会在检测时插入 V8plus 代码。因此，无论二进制文件输入如何，输出二进制文件始终为 v8plus。

可使用 discover 命令检测二进制文件。例如，以下命令将检测二进制文件 a.out，并使用检测过的 a.out 来覆盖输入 a.out：

```
discover a.out
```

当运行检测过的二进制文件时，discover 会监视程序对内存的使用。在运行过程中，discover 会将详细说明任何内存访问错误的报告写入一个 HTML 文件，您可以在 Web 浏览器中查看该文件。缺省文件名为 a.out.html。要请求将报告写入 ASCII 文件或 stderr，请在检测二进制文件时使用 -w 选项。

要指定您希望 discover 对二进制文件执行仅写入检测，请使用 -n 选项。

当 discover 检测二进制文件时，如果发现任何由于未注释而导致其无法检测的代码，将显示与以下内容类似的警告：

```
discover: (warning): a.out: 80% of code instrumented (16 out of 20 functions)
```

无注释代码可能来自链接到二进制文件中的汇编语言代码，或者来自使用早于“正确准备二进制文件” [11]中所列版本的编译器或操作系统编译的模块。

高速缓存共享库

当 discover 检测二进制文件时，会向与运行时链接程序一起工作的二进制文件中添加代码，以便在运行时装入相关的共享库时对其进行检测。检测过的库存储在高速缓存中；如果原始库自上次检测以来未发生更改，可以重新使用这些库。缺省情况下，高速缓存目录为 $HOME/SUNW_Bit_Cache。可以使用 -D 选项更改该目录。

检测共享库

如果检测整个程序（包括所有共享库），discover 实用程序会生成最准确的结果。缺省情况下，discover 仅检查并报告可执行文件中的内存错误。要指定您希望 discover 跳过对可执行文件中错误的检查，请使用 -n 选项。

可以使用 -c 选项指定希望 discover 检查相关共享库和通过 dlopen（） 动态打开的库中的错误。还可以使用 -c 选项避免检查特定的库中的错误。尽管 discover 不报告该库中的任何错误，但由于其需要跟踪整个地址空间的内存状态以正确检测内存错误，因此会记录整个程序（包括共享库）中的分配和内存初始化。

discover 实用程序运行时使用链接程序审计接口（也称为 rtld-audit 或 LD_AUDIT）从 discover 的高速缓存目录自动装入检测过的共享库。在 Oracle Solaris 上，缺省情况

下使用审计接口。在 Linux 上，当运行检测过的二进制文件时，需要在命令行上设置 LD_AUDIT。

对于 Oracle Linux 上的 32 位应用程序：

```
% LD_AUDIT=install-dir/lib/compilers/postopt/bitdl.so a.out
```

对于 Oracle Linux 上的 64 位应用程序：

```
% LD_AUDIT=install-dir/lib/compilers/postopt/amd64/bitdl.so a.out
```

该机制可能不会在所有运行 Oracle Enterprise Linux 5.x 的环境中起作用。如果不需要库检测并且未设置 LD_AUDIT，则 discover 在 Oracle Enterprise Linux 5.x 上没有问题。

应根据“正确准备二进制文件” [11]中的说明准备程序使用的所有共享库。缺省情况下，如果运行时链接程序遇到一个未准备好的库，会发生致命错误。不过，您可以指示 discover 忽略一个或多个库。

忽略库

您可能无法准备或检测某些库。可以使用 -s、-T 或 -N 选项（请参见“检测选项” [17]）或通过在 bit.rc 文件中进行指定（请参见“bit.rc 初始化文件” [19]）指示 discover 忽略这些库。可能会损失一些准确性。

如果某个库无法检测，且无法标识为可忽略，discover 将在检测时失败，或者程序将在运行时失败并出现错误消息。

缺省情况下，discover 通过在系统 bit.rc 文件中进行指定来将某些系统和编译器提供的库设置为忽略（因为没有准备这些库）。由于 discover 了解最常用库的内存特征，因此，对准确性的影响微乎其微。

检查库或可执行文件的部分

可以使用 -c 选项指定某个可执行文件或库。可以通过将内存访问检查限定至某些目标文件来进一步限定目标可执行文件或目标库。

例如，如果目标库为 libx.so 并且目标可执行文件为 a.out，则可使用以下命令：

```
$ discover -c libx.so -o a.out.disc a.out
```

还可以通过添加冒号分隔的文件或目录来限制对任何目标的检查。文件可以是 ELF 文件或目录。如果指定 ELF 文件，则会检查文件中定义的所有函数。如果指定目录，则会递归使用目录中的所有文件。

```
$ discover -o a.out.disc a.out:t1.0:dir
```

```
$ discover -c libx.so:l1.o:12.o -o a.out.disc a.out
```

命令行选项

您可以将以下选项与 `discover` 命令结合使用来检测二进制文件。

输出选项

`-a`
将错误数据写入 *binary-name*`.analyze/dynamic` 目录以供代码分析器使用。

`-b` *browser*
运行检测过的程序时会自动启动 Web 浏览器 *browser*（缺省情况下为 `off`）。

`-e` *n*
仅在报告中显示 *n* 个内存错误（缺省情况下显示所有错误）。

`-E` *n*
仅在报告中显示 *n* 个内存泄漏（缺省情况下显示 100 个）。

`-f`
在报告中显示偏移（缺省情况下隐藏偏移）。

`-H` *html-file*
将 `discover` 有关二进制文件的报告以 HTML 格式写入 *html-file*。此文件是在您运行检测过的二进制文件时创建的。如果 *html-file* 是相对路径名，则会相对于您在其中运行检测过的二进制文件的工作目录放置该文件。要使文件名在您每次运行二进制文件时都是唯一的，请将字符串 `%p` 添加到文件名中，以指示 `discover` 运行时包含进程 ID。例如，选项 `-H report.%p.html` 生成文件名为 `report.`*process-ID*`.html` 的报告文件。如果在文件名中多次包含 `%p`，则仅将第一个实例替换为进程 ID。

如果您未指定该选项或 `-w` 选项，则以 HTML 格式将报告写入 *output-file*`.html`，其中 *output-file* 是检测过的二进制文件的基名。该文件位于您运行检测过的二进制文件时所在的工作目录中。

您可以同时指定此选项和 `-w` 选项，同时以文本和 HTML 格式写入报告。

`-m`
在报告中显示改编名称（缺省为显示取消改编名称）。

`-o` *file*
将检测过的二进制文件写入 *file*。缺省情况下，检测过的二进制文件会覆盖输入二进制文件。

`-S` *n*
仅在报告中显示 *n* 个堆栈帧（缺省情况下显示 8 个）。

`-w` *text-file*
将 `discover` 有关二进制文件的报告写入 *text-file*。该文件是在您运行检测过的二进制文件时创建的。如果 *text-file* 是相对路径名，则会相对于您在其中运行检测过的二进制文件的工作目录放置该文件。要使文件名在您每次运行二进制文件时都是唯一的，请将字符串 `%p` 添加到文件名中，以指示 `discover` 运行时包含进程 ID。例

如，选项 -w report.%p.txt 生成文件名为 report.*process-ID*.txt 的报告文件。如果在文件名中多次包含 %p，则仅将第一个实例替换为进程 ID。指定 -w - 将在 stderr 中输出。

如果您未指定该选项或 -H 选项，则以 HTML 格式将报告写入 *output-file*.html，其中 *output-file* 是检测过的二进制文件的基名。该文件位于您运行检测过的二进制文件时所在的工作目录中。

您可以同时指定此选项和 -H 选项，同时以文本和 HTML 格式写入报告。

注 - 使用 -w 和 -H 选项时，建议使用完整路径名。如果使用了相对路径，则会在相对于进程运行目录的目录中生成报告。因此，如果应用程序更改了目录并启动新的进程，报告可能会放在不正确的位置。应用程序派生新进程时，libdiscoverADI.so 在运行时会为子进程创建一个父错误报告副本，子进程继续向该副本写入。如果子进程的运行目录发生变化，且对报告文件使用了相对路径，子进程可能会找不到父进程。使用完整路径名可以防止出现这些问题。

检测选项

-A [on | off]

打开/关闭分配/释放堆栈跟踪（堆栈深度为 8 时缺省值为 on）。仅当针对硬件辅助检查进行检测时，才可以使用 -i adi 选项指定此标志。要获得更好的运行时性能，可以使用此选项关闭分配/释放堆栈跟踪收集。仅当安装了 Oracle Solaris Studio 12.4 4/15 平台特定增强 (Platform Specific Enhancement, PSE) 时，才能使用此选项。

-c [-| *library [: scope...]* | *file*]

检查所有库中、指定的 *library* 中或指定的 *file* 中列出且由新行分隔的库中的错误。缺省设置为不检查库中的错误。可以通过添加冒号分隔的文件或目录来限制对库进行检查的范围。有关更多信息，请参见“检查库或可执行文件的部分” [15]。

-F [parent | child | both]

指定如果您已使用 discover 检测的二进制文件在运行时派生，您希望发生什么情况。缺省情况下，discover 继续从父进程和子进程收集内存访问错误数据。如果您希望 discover 仅跟随父进程，则指定 -F parent。如果您希望 Discover 仅跟随子进程，则指定 -F child。

-i [datarace | memcheck | adi]

确定 discover 的检测类型（缺省值为 memcheck）。

如果指定了 datarace，则使用线程分析器针对数据争用检测进行检测。如果使用此选项，仅在运行时执行数据争用检测，而不执行其他任何内存检查。必须使用 collect 命令运行检测过的二进制文件，以生成可以在性能分析器中查看的实验。有关更多信息，请参见《Oracle Solaris Studio 12.4：线程分析器用户指南》。如果指定了 memcheck，则会针对内存错误检查进行检测。如果指定了 adi，则会使用 SPARC M7 处理器的 ADI 功能针对硬件辅助检查进

行检测。只有在 SPARC M7 处理器上运行的 Oracle Solaris 11.3 中，才能使用此功能。仅当安装了 Oracle Solaris Studio 12.4 4/15 平台特定增强 (Platform Specific Enhancement, PSE) 时，才能使用 `-i adi` 选项。

`-K` 不读取 `bit.rc` 初始化文件（请参见“bit.rc 初始化文件” [19]）。

`-l` 在轻量模式下运行 `discover`。使用该选项可更快地执行程序，不必专门准备程序，但检测到的错误数会受到限制。

`-n` 不检查可执行文件中的错误。

`-N` *library* 不检测与前缀 *library* 匹配的任何相关共享库。如果库名称的前几个字符与 *library* 匹配，则忽略该库。如果 *library* 以斜杠 (*/*) 开头，则在库的完整绝对路径名上执行匹配。否则，根据库的基本名称进行匹配。

`-P [on | off]` 打开或关闭精确 ADI 模式。缺省值为 `on`。仅当针对硬件辅助检查进行检测时，才可以使用 `-i adi` 选项指定此标志。要获得更好的运行时性能，可以使用此选项关闭精确 ADI 模式。仅当安装了 Oracle Solaris Studio 12.4 4/15 平台特定增强 (Platform Specific Enhancement, PSE) 时，才能使用此选项。

`-s` 如果尝试检测不可检测的二进制文件，将会发出警告，但不标记错误。

`-T` 仅检测命名的二进制文件。在运行时不检测任何相关的共享库。

高速缓存选项

`-D` *cache-directory* 将 *cache-directory* 用作用于存储高速缓存的已检测二进制文件的根目录。缺省情况下，高速缓存目录为 `$HOME/SUNW_Bit_Cache`。

`-k` 强制重新检测高速缓存中找到的所有库。

其他选项

`-h` 或 `-?` 帮助。输出简短的用法消息并退出。

`-v` 详细。输出 `discover` 正在执行的操作的日志。指定该选项两次可获取更多信息。

`-V` 输出 `discover` 版本信息并退出。

bit.rc 初始化文件

discover 实用程序通过在启动时读取一系列 bit.rc 文件来初始化其状态。系统文件 *Oracle-Solaris-Studio-installation-directory*/prod/lib/bit.rc 提供某些变量的缺省值。discover 实用程序首先读取该文件，接着读取 $HOME/.bit.rc（如果存在），然后读取 *current-directory*/.bit.rc（如果存在）。

bit.rc 文件包含用于设置、附加或删除某些变量值的命令。当 discover 读取 set 命令时，会放弃变量以前的值（如果有）。当读取 append 命令时，会将参数附加到变量的现有值（该参数置于冒号分隔符后面）。当读取 remove 命令时，将从变量的现有值中删除参数及其冒号分隔符。

bit.rc 文件中设置的变量包括检测时要忽略的库列表，以及计算库中无注释（未准备）代码百分比时要忽略的函数或函数前缀的列表。

有关更多信息，请参阅系统 bit.rc 文件头中的注释。

运行检测过的二进制文件

使用 discover 检测二进制文件后，按照您通常使用的方法运行该二进制文件。通常，如果特定的输入组合导致您的程序行为异常，您应使用 discover 检测该程序并使用相同的输入运行该程序以调查潜在的内存问题。检测过的程序在运行时，discover 会将有关任何其发现的内存问题的信息以所选的格式（文本、HTML 或两者）写入指定的输出文件。有关解释报告的信息，请参见“分析 discover 报告” [25]。

由于检测的开销，程序可能会在您对其进行检测后运行速度明显减慢。根据内存访问的频率，运行速度最多可能会慢 50 倍。

使用芯片保护内存 (Silicon Secured Memory, SSM) 的硬件辅助检查

Oracle SPARC M7 处理器提供了软件芯片化技术，借助此技术，软件可以更快更可靠地运行。其中一种软件芯片化功能是芯片保护内存 (Silicon Secured Memory, SSM)，以前称为应用程序数据完整性 (Application Data Integrity, ADI)，其电路系统会检测常见的内存访问错误，这些错误会导致运行时数据损坏。

错误的代码或对服务器内存的恶意攻击会导致这些错误。例如，众所周知缓冲区溢出是安全漏洞的一个主要来源。内存中数据库由于在内存中有重要的数据，因而会使应用程序更容易出现此类错误。

芯片保护内存技术通过向应用程序的内存指针及其指向的内存添加版本号，避免了在优化的生产代码中出现内存损坏情况。如果指针版本号与内容版本号不匹配，内存访问就会中止。芯片保护内存技术可以与采用系统级编程语言（例如 C 或 C++）编写的应用程序一起使用，这类应用程序更容易由于软件错误而出现内存损坏。

Oracle Solaris Studio 12.4 4/15 平台特定增强 (Platform Specific Enhancement, PSE) 包括 `libdiscoverADI.so` 库（也称为 `discover` ADI 库），该库提供了更新的 `malloc`（）库例程，这些库例程可确保为相邻数据结构提供不同的版本号。借助这些版本号，处理器的 SSM 技术可以检测缓冲区溢出问题。内存结构释放后内存内容版本号会更改，以防止访问过时的指针。有关 `discover` 和 `libdiscoverADI.so` 捕获的错误的更多信息，请参见“libdiscoverADI 捕获的错误” [20]。

除了在生产环境中使用芯片保护内存技术检测潜在的内存损坏问题外，还可以在应用程序开发过程中使用该技术，从而确保在应用程序测试和认证过程中捕获此类错误。内存损坏错误是极难发现的，因为在损坏发生后要过很长时间应用程序才会遇到损坏的数据。`discover` 工具和 `libdiscoverADI.so` 库（属于 Oracle Solaris Studio 开发工具套件的一部分）提供了更多的应用程序信息，利用这些信息可以更方便地查找和修复错误的代码。

使用 libdiscoverADI 库查找内存访问错误

`discover` ADI 库 `libdiscoverADI` 会报告那些导致无效内存访问的编程错误。可以通过以下两种方式使用该库：

- 通过使用 `LD_PRELOAD_64` 环境变量将 `discover` ADI 库预装入应用程序。此方法将以 ADI 模式运行应用程序中的所有 64 位二进制文件，例如，如果正常运行一个名为 `server` 的应用程序，命令将如下所示：

  ```
  $ LD_PRELOAD_64=install-dir/lib/compilers/sparcv9/libdiscoverADI.so server
  ```

- 对特定二进制文件联合使用 ADI 模式和带 `-i adi` 选项的 `discover` 命令。

  ```
  % discover -i adi a.out
  % a.out
  ```

缺省情况下在 `a.out.html` 文件中报告错误。有关 `discover` 报告的更多信息，请参见“分析 discover 报告” [25]和“输出选项” [16]。

请参见“libdiscoverADI 的使用要求和限制” [22]。

libdiscoverADI 捕获的错误

`libdiscoverADI.so` 库会捕获以下错误：

- 数组越界访问 (ABR/ABW)
- 访问释放的内存 (FMR/FMW)
- 访问过时的指针（FMR/FMW 的一种特殊类型）
- 读取/写入未分配的内存 (UAR/UAW)
- 重复释放内存 (Double Free Memory, DFM)

有关上述每种错误类型的更多信息，请参见“内存访问错误和警告” [39]。

您的应用程序可能会管理自己的内存分配和释放列表，例如，在程序中分配大的内存块，然后对其进行细分。对于您所管理的内存，要了解如何使用 ADI 版本控制 API 捕获这类内存错误的信息，请参见Using Application Data Integrity and Oracle Solaris Studio to Find and Fix Memory Access Errors (`https://community.oracle.com/docs/DOC-912448`)。

有关完整示例，请参见“使用 `discover` ADI 模式的示例” [22]。

`discover` ADI 模式的检测选项

以下选项确定使用 ADI 模式进行检测时，在 `discover` 报告中生成的信息的精确度和数量。

`-A [on | off]`
: 将此标志设置为 `on` 时，`discover` ADI 库会报告错误的位置以及错误堆栈跟踪。此信息足以捕获错误，但不是始终都足以修复错误。此标志还会生成在何处分配和释放了违规内存区域的信息。例如，输出可能会显示错误为 `Array out of Bounds Access`（数组越界访问）以及在何处分配了此数组。如果设置为 `off`，则不会报告分配和堆栈跟踪情况。缺省值为 `on`。

注 - 由于以下原因，即使 `-A` 设置为 `on`，ABR/ABW 有时也可能会被报告为 FMR/FMW 或 UAR/UAW：

- 如果缓冲区溢出访问发生在缓冲区末尾之后或缓冲区开头之前，且距离缓冲区末尾或缓冲区开头很远。
- 如果 `libdiscoverADI.so` 达到了资源限制。在这种情况下，`discover` 可能记录了必要的分配堆栈跟踪，足以确定错误是否为缓冲区溢出。

`-P [on | off]`
: 如果将此标志设置为 `off`，ADI 将以非精确模式运行。在非精确模式下，会在确切指令执行后，再过几条指令（源代码行）捕获内存写入错误。要启用精确模式，请将此标志设置为 `on`，这是缺省值。

要获得更好的运行时性能，可以指定 `-A off` 或 `-P off`，或同时将这两个选项设置为 `off`。

`libdiscoverADI` 的使用要求和限制

`discover` 的 ADI 模式只能用于符合以下要求的 64 位应用程序：采用 SPARC M7 芯片，此芯片至少运行 Oracle Solaris 11.2.8 或 Oracle Solaris 11.3，且系统安装了 Oracle Solaris Studio 12.4 4/15 平台特定增强 (Platform Specific Enhancement, PSE)。

类似于针对内存检查的检测，如果 `libdiscoverADI.so` 的函数插入相同的分配函数，预装入的库可能发生冲突。有关更多信息，请参见“使用预装入或审计的二进制文件不兼容” [12]。

使用 `libdiscoverADI` 检查代码的其他限制如下：

- 只提供堆检查。不提供堆栈检查、静态数组越界检查和泄露检测。
- 如果应用程序使用 64 位地址中未使用的位来存储元数据，则不适用于此类应用程序。某些 64 位应用程序可能使用 64 位地址中当前未使用的高位来存储元数据（例如，锁）。此类应用程序无法与 ADI 模式下的 `discover` 一起使用，因为该功能使用 64 位地址中的 4 个最高位存储版本信息。
- 如果应用程序的指针运算有关于堆地址的假设（例如两个连续分配之间的距离），则可能不适用于此类应用程序。
- 与 `memcheck` 模式（使用 `-i memcheck` 进行检测）不同，如果应用程序在可执行文件中重新定义了标准内存分配函数，则 ADI 模式将不捕获错误。如果应用程序在库中重新定义了标准内存分配函数，则 ADI 模式会起作用。
- 缓冲区溢出的分辨度是 64 字节。对于 64 字节对齐的分配，`libdiscoverADI.so` 将按 1 字节或更多字节捕获溢出。对于非 64 位对齐的分配，报告缓冲区溢出时可能会丢失几个字节。一般情况下，不捕获 1 至 63 字节的溢出，具体取决于分配的对齐方式以及 `libdiscoverADI.so` 在高速缓存行中放置分配的位置。
- 使用 `-xipo=2` 编译的二进制文件可能具有经过内存优化的代码，这些代码处理地址的方式可能导致误报 ADI 错误，并由于陷阱处理而导致性能降级，但这种几率很小。

使用 `discover` ADI 模式的示例

本节提供了一个存在数组越界错误的代码样例，这些错误将通过 ADI 模式的 `discover` 进行捕获和报告。

假设以下样例代码位于名为 `testcode.c` 的文件中。

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

int main() {

  char *x = (char*)malloc(512);
  int *y = (int*)malloc(20*sizeof(int));
  char *z = (char*)malloc(64);
```

```
  x[-14] = 0;
  y[-10] = 0;
  z[-4] = 0;
  x[16] = 0;
  y[20] = 0;
  z[64] = 0;
  x[20] = 0;
  y[26] = 0;
  z[120] = 0;

}
```

您使用以下命令生成测试代码：

```
$ cc testcode.c -g -m64
```

要使用 ADI 模式执行此样例应用程序，请使用以下命令：

```
$ discover -w - -i adi -o a.out.adi a.out
$ ./a.out.adi
```

此命令在 discover 报告中生成以下输出。有关读取和理解这些报告的更多信息，请参见“分析 discover 报告” [25]。

```
ERROR 1 (ABW): writing to memory beyond array bounds at address 0x2fffffff7e877ff2:
        main() + 0x3c  <test-abrw.c:10>
                 7:      int *y = (int*)malloc(20*sizeof(int));
                 8:      char *z = (char*)malloc(64);
                 9:
                10:=>    x[-14] = 0;
                11:      y[-10] = 0;
                12:      z[-4] = 0;
                13:      x[16] = 0;
        _start() + 0x108
    was allocated at (512 bytes):
        main() + 0x8  <test-abrw.c:6>
                3:
                4:    int main() {
                5:
                6:=>    char *x = (char*)malloc(512);
                7:      int *y = (int*)malloc(20*sizeof(int));
                8:      char *z = (char*)malloc(64);
                9:
        _start() + 0x108
ERROR 2 (ABW): writing to memory beyond array bounds at address 0x2fffffff7e873ffc:
        main() + 0x50  <test-abrw.c:12>
                 9:
                10:      x[-14] = 0;
                11:      y[-10] = 0;
                12:=>    z[-4] = 0;
                13:      x[16] = 0;
                14:      y[20] = 0;
                15:      z[64] = 0;
        _start() + 0x108
```

```
    was allocated at (64 bytes):
        main() + 0x28  <test-abrw.c:8>
                 5:
                 6:      char *x = (char*)malloc(512);
                 7:      int *y = (int*)malloc(20*sizeof(int));
                 8:=>    char *z = (char*)malloc(64);
                 9:
                10:      x[-14] = 0;
                11:      y[-10] = 0;
        _start() + 0x108
ERROR 3 (ABW): writing to memory beyond array bounds at address 0x2fffffff7e876080:
        main() + 0x64  <test-abrw.c:14>
                11:      y[-10] = 0;
                12:      z[-4] = 0;
                13:      x[16] = 0;
                14:=>    y[20] = 0;
                15:      z[64] = 0;
                16:      x[20] = 0;
                17:      y[26] = 0;
        _start() + 0x108
    was allocated at (128 bytes):
        main() + 0x18  <test-abrw.c:7>
                 4:    int main() {
                 5:
                 6:      char *x = (char*)malloc(512);
                 7:=>    int *y = (int*)malloc(20*sizeof(int));
                 8:      char *z = (char*)malloc(64);
                 9:
                10:      x[-14] = 0;
        _start() + 0x108
ERROR 4 (ABW): writing to memory beyond array bounds at address 0x2fffffff7e874040:
        main() + 0x70  <test-abrw.c:15>
                12:      z[-4] = 0;
                13:      x[16] = 0;
                14:      y[20] = 0;
                15:=>    z[64] = 0;
                16:      x[20] = 0;
                17:      y[26] = 0;
                18:      z[120] = 0;
        _start() + 0x108
    was allocated at (64 bytes):
        main() + 0x28  <test-abrw.c:8>
                 5:
                 6:      char *x = (char*)malloc(512);
                 7:      int *y = (int*)malloc(20*sizeof(int));
                 8:=>    char *z = (char*)malloc(64);
                 9:
                10:      x[-14] = 0;
                11:      y[-10] = 0;
        _start() + 0x108
ERROR 5 (ABW): writing to memory beyond array bounds at address 0x2fffffff7e876098:
        main() + 0x84  <test-abrw.c:17>
                14:      y[20] = 0;
                15:      z[64] = 0;
```

```
                16:      x[20] = 0;
                17:=>    y[26] = 0;
                18:      z[120] = 0;
                19:
                20:    }
        _start() + 0x108
    was allocated at (128 bytes):
        main() + 0x18  <test-abrw.c:7>
                 4:    int main() {
                 5:
                 6:      char *x = (char*)malloc(512);
                 7:=>    int *y = (int*)malloc(20*sizeof(int));
                 8:      char *z = (char*)malloc(64);
                 9:
                10:      x[-14] = 0;
        _start() + 0x108
ERROR 6 (ABW): writing to memory beyond array bounds at address 0x2fffffff7e874078:
        main() + 0x90  <test-abrw.c:18>
                15:      z[64] = 0;
                16:      x[20] = 0;
                17:      y[26] = 0;
                18:=>    z[120] = 0;
                19:
                20:    }
                21:
        _start() + 0x108
    was allocated at (64 bytes):
        main() + 0x28  <test-abrw.c:8>
                 5:
                 6:      char *x = (char*)malloc(512);
                 7:      int *y = (int*)malloc(20*sizeof(int));
                 8:=>    char *z = (char*)malloc(64);
                 9:
                10:      x[-14] = 0;
                11:      y[-10] = 0;
        _start() + 0x108
DISCOVER SUMMARY:
        unique errors   : 6 (6 total)
```

分析 **discover** 报告

discover 报告为您提供用于有效精确定位并修复源代码中的问题的信息。

缺省情况下，该报告以 HTML 格式写入 *output-file*.html，其中 *output-file* 是检测过的二进制文件的基名。该文件位于您运行检测过的二进制文件时所在的工作目录中。

检测二进制文件时，可以使用 -H 选项请求将 HTML 输出写入指定的文件，或使用 -w 选项请求将其写入文本文件。

检测二进制文件后，如果您希望将报告写入其他文件以便后续运行该程序，则可以通过 SUNW_DISCOVER_OPTIONS 环境变量为报告更改 -H 和 -w 选项的设置。有关更多信息，请参见“SUNW_DISCOVER_OPTIONS 环境变量” [38]。

注 - 如果在检测代码时指定了 -a 选项，则必须使用代码分析器或 codean 命令读取报告。

分析 HTML 报告

使用 HTML 报告格式可以对程序进行交互分析。开发者可以通过使用电子邮件或者通过发布到 Web 页上来轻松共享 HTML 格式的数据。该格式与 JavaScript 交互式功能相结合，提供了一种在 discover 消息中导航的便捷方法。

本节介绍 HTML 报告，其中包含以下选项卡：

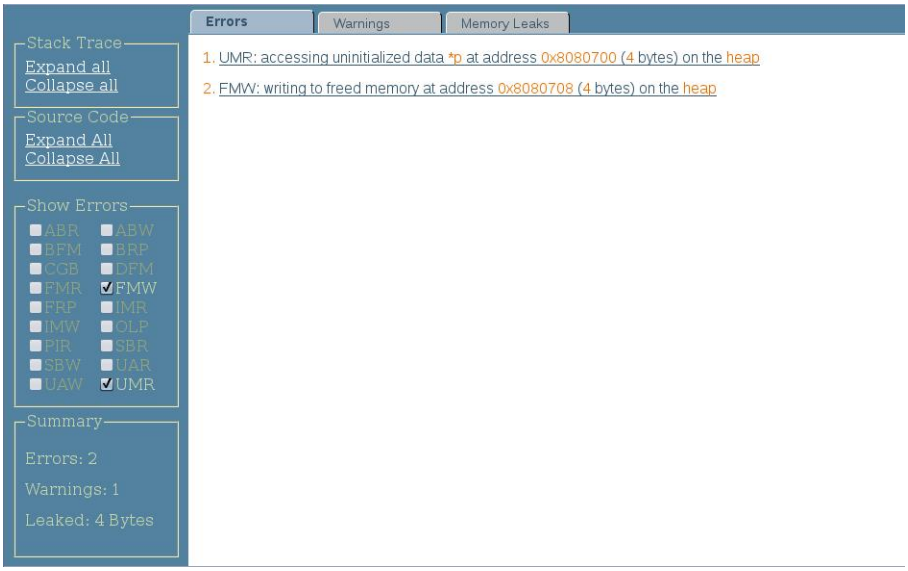
- “使用 "Errors"（错误）选项卡” [26]
- “使用 "Warnings"（警告）选项卡” [28]
- “使用 "Memory Leaks"（内存泄漏）选项卡” [29]

使用 "Errors"（错误）选项卡、"Warnings"（警告）选项卡和 "Memory Leaks"（内存泄漏）选项卡，可以分别在错误消息、警告消息和内存泄漏报告中导航。

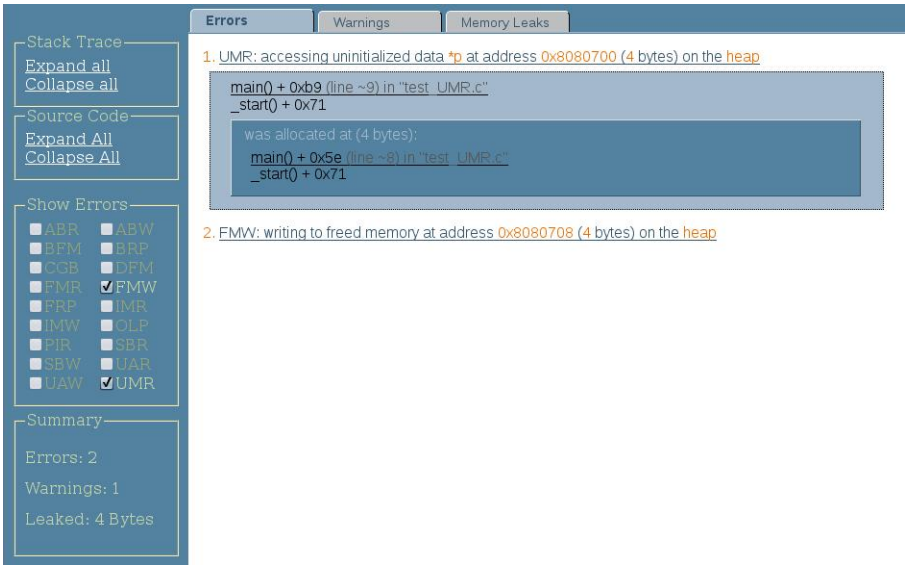
使用左侧的控制面板，可以更改当前显示在右侧的选项卡的内容。请参见“使用控制面板” [30]。

使用 "Errors"（错误）选项卡

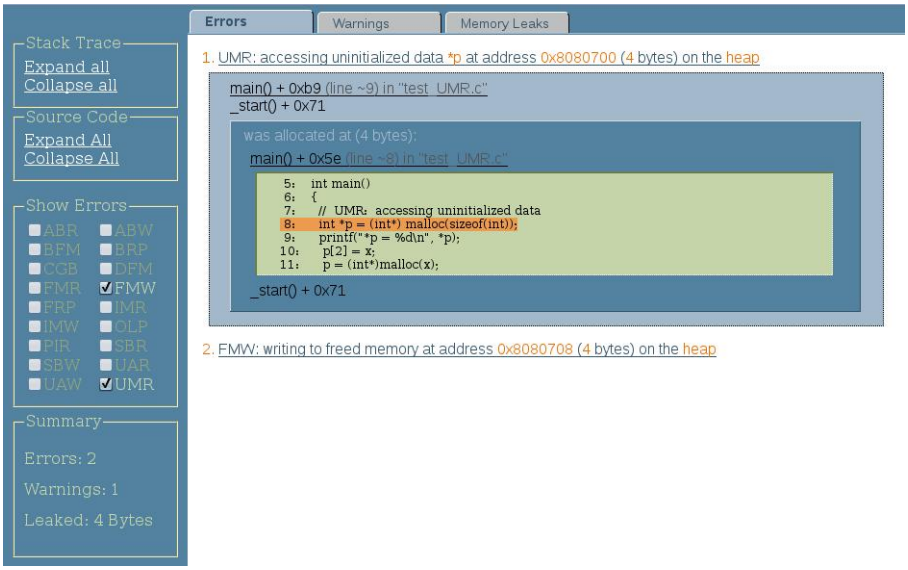
在浏览器中第一次打开 HTML 报告时，会选择 "Errors"（错误）选项卡，其中显示在检测过的二进制文件执行过程中发生的内存访问错误列表：



单击某个错误时，会显示发生该错误时的堆栈跟踪：

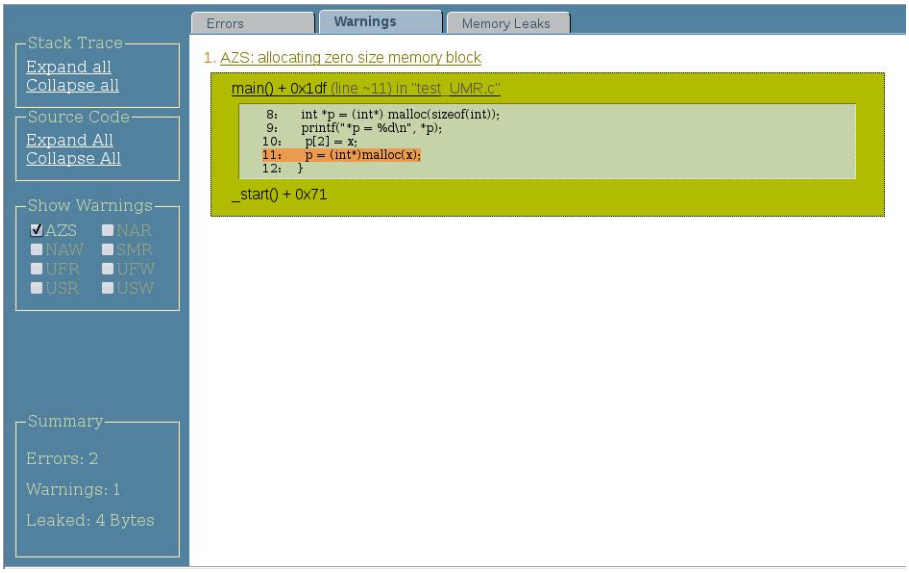


如果编译代码时使用了 -g 选项，则可以通过单击堆栈跟踪中的每个函数来查看相应函数的源代码：



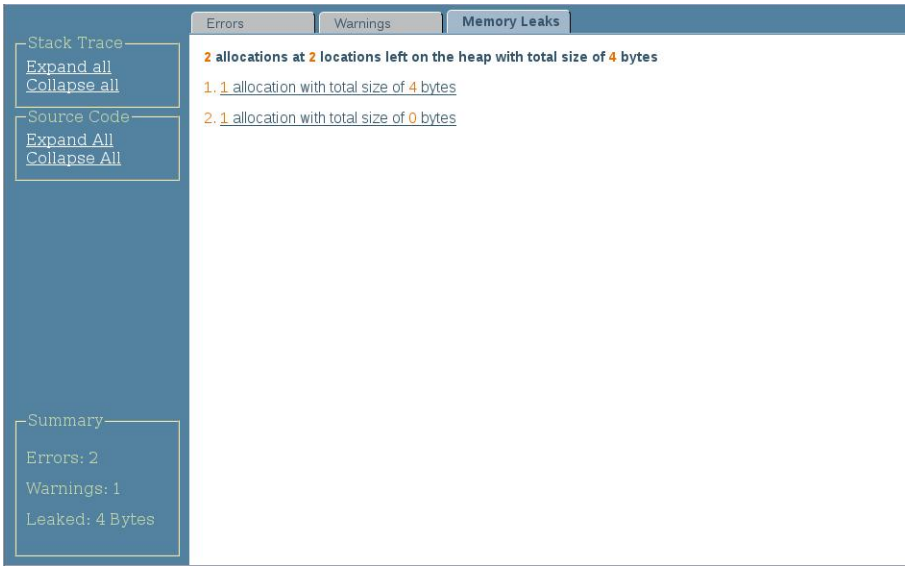
使用 "Warnings"（警告）选项卡

"Warnings"（警告）选项卡显示了有关可能的访问错误的所有警告消息。单击某一条警告时，会显示发出该警告时的堆栈跟踪。如果编译代码时使用了 -g 选项，则可以通过单击堆栈跟踪中的每个函数来查看相应函数的源代码：

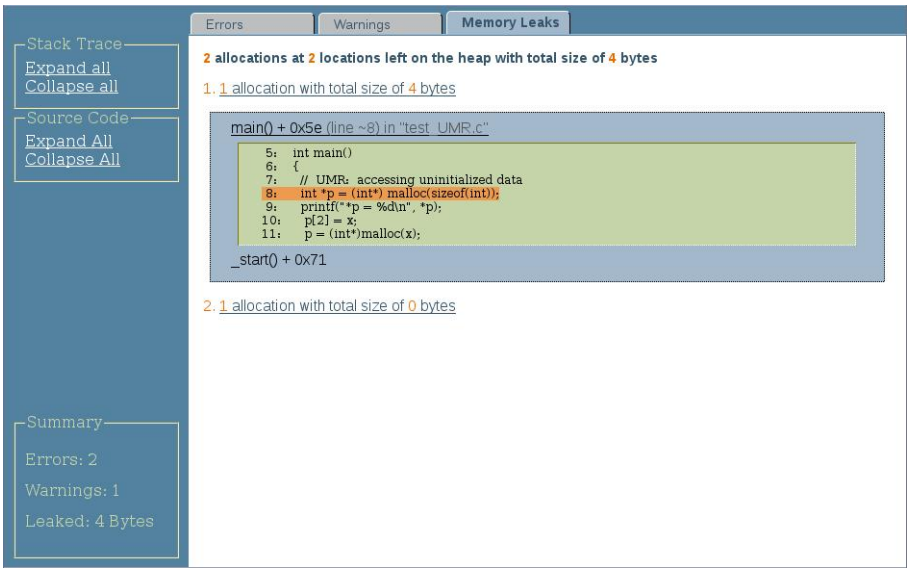


使用 "Memory Leaks"（内存泄漏）选项卡

"Memory Leaks"（内存泄漏）选项卡在顶部显示在程序运行结束时仍被分配的块的总数，并在下方列出这些块：



单击某个块时，会显示该块的堆栈跟踪。如果编译代码时使用了 -g 选项，则可以通过单击堆栈跟踪中的每个函数来查看相应函数的源代码：



使用控制面板

要查看所有错误、警告和内存泄漏的堆栈跟踪，请在控制面板的 "Stack Traces"（堆栈跟踪）部分中单击 "Expand All"（全部展开）。要查看所有函数的源代码，请在控制面板的 "Source Code"（源代码）部分中单击 "Expand All"（全部展开）。

要隐藏所有错误、警告和内存泄漏的堆栈跟踪或源代码，请单击相应的 "Collapse All"（全部折叠）。

选择相关的选项卡时，会显示控制面板的 "Show Errors"（显示错误）或 "Show Warnings"（显示警告）部分。缺省情况下，会选中所有检测到的错误或警告的选项。要隐藏某种类型的错误或警告，请将其取消选中。

控制面板的底部显示了报告摘要，其中列出了错误和警告总数，以及泄漏的内存量。

分析 ASCII 报告

discover 报告的 ASCII（文本）格式适合由脚本进行处理或无法访问 Web 浏览器的情况。以下示例显示了一个 ASCII 报告样例。

```
$ a.out

ERROR 1 (UAW): writing to unallocated memory at address 0x50088 (4 bytes) at:
main() + 0x2a0  <ui.c:20>
17:     t = malloc(32);
18:     printf("hello\n");
19:     for (int i=0; i<100;i++)
20:=>    t[32] = 234; // UAW
21:     printf("%d\n", t[2]);   //UMR
22:     foo();
23:     bar();
_start() + 0x108
ERROR 2 (UMR): accessing uninitialized data from address 0x50010 (4 bytes) at:
main() + 0x16c  <ui.c:21>$
18:     printf("hello\n");
19:     for (int i=0; i<100;i++)
20:      t[32] = 234; // UAW
21:=>   printf("%d\n", t[2]);   //UMR
22:     foo();
23:     bar();
24:    }
_start() + 0x108
was allocated at (32 bytes):
main() + 0x24  <ui.c:17>
14:     x = (int*)malloc(size); // AZS warning
15:    }
16:    int main() {
17:=>   t = malloc(32);
18:     printf("hello\n");
19:     for (int i=0; i<100;i++)
20:      t[32] = 234; // UAW
_start() + 0x108
0
WARNING 1 (AZS): allocating zero size memory block at:
foo() + 0xf4  <ui.c:14>
11:    void foo() {
12:     x = malloc(128);
13:     free(x);
14:=>   x = (int*)malloc(size); // AZS warning
15:    }
16:    int main() {
17:     t = malloc(32);
main() + 0x18c  <ui.c:22>
19:     for (int i=0; i<100;i++)
20:      t[32] = 234; // UAW
21:     printf("%d\n", t[2]);   //UMR
22:=>   foo();
23:     bar();
24:    }
_start() + 0x108

***************** Discover Memory Report *****************

1 block at 1 location left allocated on heap with a total size of 128 bytes
```

```
1 block with total size of 128 bytes
bar() + 0x24  <ui.c:9>
6:           7:    void bar() {
8:     int *y;
9:=>   y = malloc(128);  // Memory leak
10:    }
11:    void foo() {
12:     x = malloc(128);
main() + 0x194  <ui.c:23>
20:      t[32] = 234; // UAW
21:     printf("%d\n", t[2]);   //UMR
22:     foo();
23:=>   bar();
24:    }
_start() + 0x108

ERROR 1: repeats 100 times
DISCOVER SUMMARY:
unique errors   : 2 (101 total, 0 filtered)
unique warnings : 1 (1 total, 0 filtered)
```

该报告包括错误和警告消息，其后是摘要。

ASCII 警告和错误消息说明

错误消息以单词 ERROR 开头并包含由三个字母组成的代码、ID 号和错误说明（本例中为 writing to unallocated memory）。其他详细信息包括访问的内存地址和读取或写入的字节数。说明之后是发生错误时的堆栈跟踪，可精确定位进程生命周期中错误的位置。

如果使用 -g 选项编译了程序，堆栈跟踪将包含源文件名和行号。如果源文件可访问，则输出错误位置附近的源代码。每一帧中的目标源代码行由 ⇒ 符号指示。

如果同一内存位置重复出现字节数相同的同一错误类型，完整的消息（包括堆栈跟踪）仅输出一次。对于多次出现的每个相同错误，会统计后续的错误，并在报告的末尾列出重复计数（如下例中所示）。

```
ERROR 1: repeats 100 times
```

如果出错内存访问的地址位于堆上，则在堆栈跟踪的后面输出相应堆块的信息。这些信息包括块的起始地址和大小，以及分配该块时的堆栈跟踪。如果已释放该块，则报告包含取消分配点的堆栈跟踪。

警告消息的显示格式与错误消息相同，只是警告消息以单词 WARNING 开头。一般而言，这些消息向您通知的是一些不影响应用程序功能但提供了可用于改进程序的有用信息的情况。例如，分配大小为零的内存不会带来危害，但如果这种情况经常发生，就可能会使性能降级。

ASCII 内存泄漏报告

内存泄漏报告包含有关在堆上分配的、但程序退出时未释放的内存块的信息。以下示例显示了一个内存泄漏报告样例。

```
$ DISCOVER_MEMORY_LEAKS=1 ./a.out
...
***************** Discover Memory Report *****************

2 blocks left allocated on heap with total size of 44 bytes
block at 0x50008 (40 bytes long) was allocated at:
malloc() + 0x168 [libdiscover.so:0xea54]
f() + 0x1c [a.out:0x3001c]
<discover_example.c:9>:
8:    {
9:=>    int *a = (int *)malloc( n * sizeof(int) );
10:      int i, j, k;
main() + 0x1c [a.out:0x304a8]
<discover_example.c:33>:
32:      /* Print first N=10 Fibonacci numbers */
33:=>    a = f(N);
34:      printf("First %d Fibonacci numbers:\n", N);
_start() + 0x5c [a.out:0x105a8]
...
```

标题下面的第一行汇总了在堆上保持已分配状态的堆块数及其总大小。报告的大小是从开发者的立场提供的，也就是说，不包括内存分配器的簿记系统开销。

ASCII 堆栈跟踪报告

在内存泄漏摘要之后，提供有关每个未释放的堆块的详细信息及其分配点的堆栈跟踪。该堆栈跟踪报告类似于前面介绍错误和警告消息时提到的堆栈跟踪报告。

ASCII 报告摘要

discover 报告的结尾是总摘要。该摘要报告了唯一警告和错误的数目，并在括号中提供了错误和警告的总数（包括重复项）。例如：

```
DISCOVER SUMMARY:
unique errors   : 3 (3 total)
unique warnings : 1 (5 total)
```

discover API 和环境变量

有多个可供在代码中指定的 discover API 和环境变量。

discover API

Oracle Solaris Studio 12.4 实现了六个新的可以从程序调用以接收内存泄漏和内存分配信息的 discover 函数。这些函数将信息输出到 stderr。缺省情况下，discover 会在程序输出的末尾输出包含程序中内存泄漏的最终内存报告。要使用这些 API，应用程序的源文件需要包含 discover 的头文件：#include <discoverAPI.h>。

这些函数及其报告的内容如下所示：

`discover_report_all_inuse ()`
: 报告所有内存分配。

`discover_report_unreported_inuse ()`
: 报告以前未报告的所有内存分配。

`discover_mark_all_inuse_as_reported ()`
: 将迄今为止的所有内存分配标记为已报告。

`discover_report_all_leaks ()`
: 报告所有内存泄漏。

`discover_report_unreported_leaks ()`
: 报告以前未报告的所有内存泄漏。

`discover_mark_all_leaks_as_reported ()`
: 将迄今为止的所有内存泄漏标记为已报告。

本节介绍一些使用 discover API 的方法。

注 - discover API 在 ADI 模式下不起作用。

使用 discover API 查找内存泄漏

对于在您的代码中指定的每个函数，discover 会报告内存分配位置的堆栈。内存泄漏是在程序中无法访问的已分配内存。

以下示例显示如何使用这些 API：

```
$ cat -n tdata.C
     1    #include <discoverAPI.h>
     2
     3    void foo()
```

```
     4    {
     5      int *j = new int;
     6    }
     7
     8    int main()
     9    {
    10      foo();
    11      discover_report_all_leaks();
    12
    13      foo();
    14      discover_report_unreported_leaks();
    15
    16      return 0;
    17    }
$ CC -g tdata.C
$ discover -w - a.out
$ a.out
```

以下示例显示了预期输出。

```
******** discover_report_all_leaks() Report ********
1 allocation at 1 location left on the heap with a total size of 4 bytes
LEAK 1: 1 allocation with total size of 4 bytes
void*operator new(unsigned) + 0x36
void foo() + 0x5e  <tdata.C:5>
2:
3:    void foo()
4:    {
5:=>    int *j = new int;
6:    }
7:
8:    int main()
main()+0x1a  <tdata.C:10>
9:    {
10:=>    foo();
11:      discover_report_all_leaks();
12:
13:      foo();         _start() +

************************************************************
******** discover_report_unreported_leaks() Report ********
1 allocation at 1 location left on the heap with a total size of 4 bytes
LEAK 1: 1 allocation with total size of 4 bytes
void*operator new(unsigned) + 0x36
void foo() + 0x5e  <tdata.C:5>
2:
3:    void foo()
4:    {
5:=>    int *j = new int;
6:    }
7:
8:int main()
main() + 0x24  <tdata.C:13>
```

```
10:      foo();
11:      discover_report_all_leaks();
12:
13:=>    foo();
14:      discover_report_unreported_leaks();
15:
16:return 0;
_start() + 0x71

************************************************************
***************** Discover Memory Report *****************
2 allocations at 2 locations left on the heap with a total size of 8     bytes
LEAK 1: 1 allocation with total size of 4 bytes
void*operator new(unsigned) + 0x36
void foo() + 0x5e  <tdata.C:5>
2:
3:    void foo()
4:    {
5:=>    int *j = new int;
6:    }
7:
8:    int main()
main() +    0x1a  <tdata.C:10>
7:
8:    int main()
9:    {             10:=>    foo();
11:      discover_report_all_leaks();
12:
13:      foo();         _start() + 0x71
LEAK 2: 1 allocation with total size of 4 bytes
void*operator new(unsigned) + 0x36
void foo() + 0x5e  <tdata.C:5>
2:
3:    void foo()
4:    {
5:=>    int *j = new int;
6:    }
7:
8:    int main()
main() + 0x24  <tdata.C:13>
10:      foo();
11:      discover_report_all_leaks();
12:
13:=>    foo();
14:         discover_report_unreported_leaks();
15:
16:      return 0;
_start() + 0x71

DISCOVER SUMMARY:
unique errors   : 0 (0 total)
unique warnings : 0 (0 total)
```

在服务器或长期运行的程序中查找泄漏

如果您有长期运行的程序或从不退出的服务器，则可以随时使用 dbx 调用这些 discover 函数，即使您未将调用放置在代码中也可以执行此操作。必须至少已使用 -l 选项以 discover 的轻量模式运行了该程序。请注意，dbx 可以附加到正在运行的程序。以下示例显示如何在长期运行的程序中查找泄漏

例 1 在长期运行的程序中找到两处泄漏

对于本示例，a.out 文件是长期运行的具有两个进程的程序，每个进程存在一处泄漏。为每个进程分配了一个进程 ID。

以下 rl 脚本包含用于指示程序报告尚未报告的内存泄漏的命令。

```
#!/bin/sh
dbx - $1 > /dev/null 2> &1 << END
call discover_report_unreported_leaks()
exit
END
```

一旦您具有程序和脚本，就可以使用 discover 并运行该程序。

```
% discover -l -w - a.out
% a.out
8252: Parent allocation 64
8253: Child allocation 32
```

在单独的终端窗口中，可以对父进程运行脚本。

```
% rl 8252
```

程序针对父进程报告以下信息：

```
******** discover_report_unreported_leaks() Report ********

1 allocation at 1 location left on the heap with a total size of 64 bytes

    LEAK 1: 1 allocation with total size of 64 bytes
    main() + 0x1e  <xx.c:17>
        14:
        15:      if (child > 0) {
        16:
        17:=>      void *p = malloc(64);
        18:        printf("%jd: Parent allocation 64\n", (intmax_t)getpid());
        19:        p = 0;
        20:        for (int j=0; j < 1000; j++) sleep(1);
    _start() + 0x66
```

```
************************************************************
```

针对子进程再次运行脚本。

```
% rl 8253
```

程序针对子进程报告以下信息：

```
******** discover_report_unreported_leaks() Report ********

1 allocation at 1 location left on the heap with a total size of 32 bytes

    LEAK 1: 1 allocation with total size of 32 bytes
    main() + 0x80  <xx.c:24>
        21:      }
        22:
        23:      else {
        24:=>      void *p = malloc(32);
        25:        printf("%jd: Child allocation 32\n", (intmax_t)getpid());
        26:        p = 0;
        27:        for (int j=0; j < 1000; j++) sleep(1);
    _start() + 0x66


************************************************************
```

可以重复使用脚本查找任何新的泄漏。

SUNW_DISCOVER_OPTIONS 环境变量

您可以通过将 SUNW_DISCOVER_OPTIONS 环境变量设置为命令行选项 -a、-A、-b、-e、-E、-f、-F、-H、-l、-L、-m、-P、-S 和 -w 的列表来更改检测过的二进制文件的运行时行为。例如，如果您希望将报告的错误数更改为 50 并将报告中的堆栈深度限制为 3，则可以按以下方式设置环境变量：

```
-e 50 -s 3
```

SUNW_DISCOVER_FOLLOW_FORK_MODE 环境变量

缺省情况下，如果您使用 discover 检测的二进制文件在您运行该文件时派生，discover 会继续从父进程和子进程收集内存访问错误数据，这表示缺省行为是 both。例如，如果您希望 discover 跟随派生并从子进程收集内存访问数据，请将 SUNW_DISCOVER_FOLLOW_FORK_MODE 环境变量设置为：

```
-F child
```

内存访问错误和警告

discover 实用程序检测并报告大量内存访问错误，并就可能是错误的访问向您发出警告。

内存访问错误

discover 可检测到以下内存访问错误：

- 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：损坏的数组保护块
- 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
- 内存泄漏
- OLP：重叠源和目标
- PIR：部分初始化的读取
- SBR：堆栈框架越界读
- SBW：堆栈框架越界写
- 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

以下各节列出了一些将生成其中部分错误的简单样例程序。

ABR

```
  //  ABR: reading memory beyond array bounds at address 0x%1x (%d byte%s)
int *a = (int*) malloc(sizeof(int[5]));
printf("a[5] = %d\n",a[5]);
```

discover 实用程序还检测静态类型的 ABR 错误。

```
    int globalarray[5];

   int main(){
```

```
    int i, j;
    for(i = 0; i < 7; i++) {
      j = globalarray[i-1];   // Reading memory beyond static/global array bounds
    }
    return 0;
  }
```

ABW

```
  //  ABW:  writing to memory beyond array bounds
int *a = (int*) malloc(sizeof(int[5]));
a[5] = 5;
```

discover 实用程序还检测静态类型的 ABW 错误。

```
int globalarray[5];

   int main(){
     int i;
     for(i = 0; i < 7; i++) {
       globalarray[i-1] = i;   // Writing to memory beyond static/global array bounds
     }
     return 0;
   }
```

BFM

```
  //  BFM:  freeing wrong memory block
int *p = (int*) malloc(sizeof(int));
free(p+1);
```

BRP

```
  //  BRP:  bad address parameter for realloc 0x%1x
int *p = (int*) realloc(0,sizeof(int));
int *q = (int*) realloc(p+20,sizeof(int[2]));
```

CGB

```
// CGB: writing past the end of a dynamically allocated array, or being in the "red zone".
    #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;
  }
```

DFM

```
  // DFM: double freeing memory
int *p = (int*) malloc(sizeof(int));
free(p);
free(p);'
```

FMR

```
  //  FMR: reading from freed memory at address 0x%1x  (%d byte%s)
int *p = (int*) malloc(sizeof(int));
free(p);
printf("p = 0x%h\n",p);
```

FMW

```
  //  FMW: writing to freed memory at address 0x%1x (%d byte%s)
int *p = (int*) malloc(sizeof(int));
free(p);
*p = 1;
```

FRP

```
  //  FRP:  freed pointer passed to realloc
int *p = (int*) malloc(sizeof(int));
free(0);
int *q = (int*) realloc(p,sizeof(int[2]));
```

IMR

```
  //  IMR:  read from invalid memory address
int *p = 0;
int i = *p;   // generates Signal 11...
```

IMW

```
  //  IMW:  write to invalid memory address
int *p = 0;
*p = 1;       // generates Signal 11...
```

内存泄漏

```
//Memory Leak: memory allocated but not freed before exit or escaping from the function
       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
```

OLP

```
  //  OLP: source and destination overlap
char *s=(char *) malloc(15);
memset(s, 'x', 15);
memcpy(s, s+5, 10);
return 0;
```

PIR

```
  //  PIR:  accessing partially initialized data
int *p = (int*) malloc(sizeof(int));
*((char*)p) = 'c';
printf("*(p = %d\n",*(p+1));
```

SBR

```
  //  SBR: reading beyond stack frame bounds
int a[2]={0,1};
printf("a[-10]=%d\n",a[-10]);
return 0;
```

SBW

```
  //  SBW: writing beyond stack frame bounds
int a[2]={0,1)'
a[-10]=2;
return 0;
```

UAR

```
  //  UAR" reading from unallocated memory
```

```
int *p = (int*) malloc(sizeof(int));
printf("*(p+1) = %d\n",*(p+1));
```

UMR

```
  // UMR: accessing uninitialized data from address 0x%1x (A%d byte%s)
int *p = (int*) malloc(sizeof(int));
printf("*p = %d\n",*p);
```

内存访问警告

discover 实用程序会报告下列内存访问警告：

- AZS：分配零大小
- SMR：推测性未初始化内存读取

以下示例显示一个将生成 AZS 警告的简单程序。

```
  //  AZS:  allocating zero size memory block
int *p = malloc();
```

解释 discover 错误消息

在某些情况下，discover 可能报告实际上不是错误的错误。此类情况称为误报。discover 实用程序在检测时分析代码，与类似工具相比可以减少误报的发生，但在某些情况下误报仍可能发生。本节提供一些提示，可能有助于您识别并可能避免 discover 报告中的误报。

部分初始化内存

可以在 C 和 C++ 中使用位字段创建紧凑数据类型。例如：

```
struct my_struct {
unsigned int valid : 1;
char         c;
};
```

在本例中，结构成员 my_struct.valid 在内存中仅占用一位。但是，在 SPARC 平台上，CPU 只能以字节为单位修改内存，因此，只有装入含有 struct.valid 的整个字节才能访问或修改该结构成员。此外，编译器有时可能会一次装入多个字节（例如，由四

个字节构成的计算机字）。当 discover 检测到此类装入且没有其他信息时，假设会使用全部四个字节。例如，如果字段 my_struct.valid 已初始化，但字段 my_struct.c 未初始化，并且已装入包含这两个字段的计算机字，则 discover 会标记部分初始化内存读取 (PIR)。

误报的另一个原因是位字段初始化。要写入某个字节的一部分，编译器必须首先生成用于装入该字节的代码。如果该字节不是在读取之前写入的，将生成未初始化内存读取 (UMR) 错误。

要避免位字段误报，请在编译时使用 -g 选项或 -g0 选项。这些选项向 discover 提供额外的调试信息，以帮助其识别位字段装入和初始化，这将消除大多数误报。如果出于某种原因无法使用 -g 选项进行编译，请使用 memset（） 等函数初始化结构。例如：

```
...
struct my_struct s;
/* Initialize structure prior to use */
memset(&sm 0, sizeof(struct my_struct));
...
```

可疑装入

有时，当装入的结果对某些程序路径无效时，编译器会从已知的内存地址生成装入。这种情况经常发生在 SPARC 平台上，因为此类装入指令可以放置在分支指令的延迟槽中。例如，请看以下 C 语言代码片段：

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

根据此代码，编译器可能会生成与以下示例等效的代码：

```
int i;
int t1, t2'
t1 = foo(&i);
t2 = i; /* value in i is loaded */
if (t1 != 0) {
printf("%d\n", t2);
}
```

假定本例中的函数 foo（） 返回了 0 且未初始化 i。仍生成来自 i 的装入，即使并未使用它时也是如此。不过，由于 discover 会看到该装入，因此将报告未初始化的变量装入 (UMR)。

discover 实用程序会尽量使用数据流分析来识别此类情况，但有时无法检测到此类情况。

使用较低的优化级别进行编译可以减少这些类型的误报的发生。

未检测的代码

discover 有时无法检测整个程序，尤其是当部分代码来自汇编语言源文件或无法重新编译的第三方库，从而无法进行检测时。discover 无法检测未检测的代码正在访问或修改的内存块。例如，假定某个第三方共享库中的某个函数初始化了一个内存块，主程序（检测过的程序）稍后读取了该内存块。由于 discover 无法检测到该库已初始化内存，因此后续读取操作将生成未初始化内存错误 (UMR)。

为解决此类问题，discover API 中包含了下列函数：

```
void __ped_memory_write(unsigned long addr, long size, unsigned long pc);
void __ped_memory_read(unsigned long addr, long size, unsigned long pc);
void __ped_memory_copy(unsigned long src, unsigned lond dst, long size, unsigned long pc);
```

您可以从程序调用这些 API 函数，以便将特定事件（如向内存区写入 (__ped_memory_write（）) 或从内存区读取 (__ped_memory read（）)）告知 discover。对于这两种情况，内存区的起始地址将通过 addr 参数传递，内存区的大小通过 size 参数传递。将 pc 参数设置为 0。

使用 __ped_memory_copy 函数向 discover 通知正从一个位置复制到另一个位置的内存。源内存的起始地址将通过 src 参数传递，目标区的起始地址通过 dst 参数传递，大小通过 size 参数传递。将 pc 参数设置为 0。

要使用 API，请在程序中将这些函数声明为弱函数。例如，在源代码中包含以下代码片段。

```
#ifdef __cplusplus
extern "C" {
#endif

extern void __ped_memory_write(unsigned long addr, long size, unsigned long pc);
extern void __ped_memory_read(unsigned long addr, long size, unsigned long pc);
extern void __ped_memory_copy(unsigned long src, unsigned long dst, long size, unsigned long
 pc);

#prgama weak __ped_memory_write
#pragma weak __ped_memory_read
#pragma weak __ped_memory_copy

#ifdef __cplusplus
}
#endif
```

内部 discover 库（该库在检测时与您的程序链接）定义 API 函数。但是，如果未检测程序，则不会链接该库，这样，对 API 函数的所有调用都会导致应用程序挂起。因此，如果不是在 discover 下运行程序，则必须禁用这些函数。另外，您也可以使用 API 函数的空定义创建一个动态库，并将其链接到程序。在此情况下，如果您未在 discover 下运行程序，将使用您的库，但如果在 discover 下运行程序，将自动调用实际的 API 函数。

使用 discover 时的限制

本节介绍在使用 discover 时的一些已知限制。

仅检测有注释的代码

discover 实用程序只能检测按照“正确准备二进制文件” [11]中的说明准备的代码。无注释代码可能来自链接到二进制文件的汇编语言代码，或者来自使用早于本节所列版本的编译器或操作系统编译的模块。

discover 实用程序无法检测汇编语言模块或包含 asm 语句或 .il 模板的函数。

此外，Oracle Solaris Studio 12.4 C++ 运行时库不包含注释数据，因为它们未使用 Oracle Solaris Studio 编译器进行编译。如果要将 discover 用于使用 C++ 编译器生成（使用 -std=c++11 选项）的程序，discover 将不会捕获 UMR 或 PIR 错误。

计算机指令可能不同于源代码

discover 处理的是计算机代码。该工具可检测到装入和存储等计算机指令中的错误，并将这些错误与源代码相关联。由于某些源代码语句没有关联的计算机指令，因此 discover 似乎无法检测到明显的用户错误。例如，请看以下 C 语言代码片段：

```
  int *p = (int *)malloc(sizeof(int));
int i;

i = *p;  /* compiler may not generate code for this statement */
printf("Hello World!\n");

return;
```

由于内存未初始化，读取 p 所指向的地址处存储的值就是一个潜在的用户错误。但是，优化编译器可以检测到未使用变量 i，因此，不会生成让语句读取内存并将其分配给 i 的代码。在此情况下，discover 不会报告使用了未初始化内存 (UMR)。

编译器选项影响生成的代码

编译器生成的代码是不可预测的。由于编译器生成的代码会因所使用的编译器选项（包括 -O*n* 优化选项）的不同而异，因此，discover 报告的错误也可能有所不同。例如，在 -O1 优化级别上生成的代码中报告的错误可能不适用于在 -O4 优化级别上生成的代码。

如果使用编译器生成程序时使用了 C++11 标准选项，则 discover 工具无法检测 UMR 或 PIR 错误。

系统库可能会影响报告的错误

系统库是随操作系统一起预装的，无法重新编译以进行检测。discover 实用程序为标准 C 库 (libc.so) 中的公用函数提供支持；也就是说，discover 知道这些函数访问或修改的内存。但是，如果应用程序使用了其他系统库，可能会在 discover 报告中出现误报。如果出现误报，可以从代码调用 discover API 来消除误报。

定制内存管理可能会影响数据的准确性

discover 实用程序可以跟踪标准编程语言机制（如 malloc（）、calloc（）、free（）、operator new（） 和 operator delete（））分配的堆内存。

如果应用程序使用具有标准函数的定制内存管理系统（例如，使用 malloc（） 实现的池分配管理），则 discover 可能无法保证正确报告泄漏或对已释放内存的访问。

discover 实用程序不支持下列内存分配器：

- 直接使用 brk(2)（） 或 sbrk(2)（） 系统调用的定制堆分配器
- 静态链接到二进制文件中的标准堆管理函数
- 使用 mmap(2)（） 和 shmget(2)（） 系统调用从用户代码中分配的内存

不支持 signaltstack(2)（） 函数。

无法检测到静态和自动数组的超出边界错误

由于 discover 检测数组边界所用的算法问题，它无法检测到静态和自动（本地）数组自动超出边界的访问错误。但是，discover 可以检测到静态数组越界访问错误。可以检测到动态分配的数组的错误。

♦ ♦ ♦ 第 3 章

3 代码覆盖工具 (uncover)

代码覆盖工具 (uncover) 软件测量应用程序的代码覆盖。本章介绍了以下相关主题：

使用 uncover 的要求

uncover 实用程序适用于至少使用 Sun Studio 12 Update 1、Oracle Solaris Studio 12.2、Oracle Solaris Studio 12.3 编译器或 Oracle Solaris Studio 12.4 编译的二进制文件。它可以在基于 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。

按照说明进行编译的二进制文件包含一些信息，uncover 可使用这些信息可靠地反汇编该二进制文件，以便对其进行检测以收集覆盖数据。

要使 Uncover 能够使用源代码级别的覆盖信息，请在编译二进制文件时使用 -g 选项生成调试信息。如果二进制文件不是使用 -g 选项编译的，Uncover 只能使用基于程序计数器 (program counter, PC) 的覆盖信息。

uncover 实用程序可与使用 Oracle Solaris Studio 编译器生成的任何二进制文件结合使用，但与生成时不使用任何优化选项的二进制文件结合使用效果最好。以前的 uncover 发行版至少需要 -O1 优化级别。如果生成二进制文件时使用了优化选项，则 uncover 结果在使用较低优化级别（-O1 或 -O2）时较好。uncover 使用通过 -g 选项生成二进制文件时生成的调试信息将指令与行号相关联，由此派生源代码行级覆盖。优化级别为 -O3 和更高级别时，编译器可能会删除可能从不不执行的或冗余的某一代码，这可能导致没有任何二进制文件指令用于某些源代码行。在此类情况下，不会为这些行报告任何覆盖信息。有关更多信息，请参见“使用 uncover 时的限制” [63]。

使用 uncover

使用 Uncover 生成覆盖信息的过程分为三个步骤：

1. “检测二进制文件” [50]
2. “运行检测过的二进制文件” [51]
3. “生成并查看覆盖报告” [51]

本节介绍了三个步骤，并提供了使用 Uncover 的示例。

检测二进制文件

输入的二进制文件可以是可执行文件或共享库。必须分别检测要分析的每个二进制文件。

使用 `uncover` 命令检测二进制文件。例如，以下命令将检测二进制文件 `a.out`，并使用检测过的 `a.out` 来覆盖输入 `a.out`。该命令还将创建一个后缀为 `.uc` 的目录（本例中为 `a.out.uc`），将在该目录中收集覆盖数据。输入二进制文件的副本保存在此目录中。

```
$ uncover a.out
```

检测二进制文件时，可以使用以下选项：

`-c`	启用指令、块和函数的执行计数报告。缺省情况下，仅报告已覆盖或未覆盖的代码的信息。检测二进制文件和生成覆盖报告时，均指定该选项。
`-d` *directory*	在 *directory* 中创建覆盖数据目录。当您为多个二进制文件收集覆盖数据时，此选项十分有用，因为所有覆盖数据目录都是在同一个目录中创建的。此外，如果从不同的位置运行同一个检测过的二进制文件的不同实例，使用此选项可确保在同一个覆盖数据目录中累积所有这些运行中的覆盖数据。 如果不使用 `-d` 选项，将在当前运行目录中创建覆盖数据目录。
`-m on | off`	启用或禁用线程安全分析。缺省值为 `on`。将该选项与 `-c` 运行时选项结合使用。如果使用 `-m off` 检测使用线程的二进制文件，则该二进制文件会在运行时失败，并且显示一条消息，要求您使用 `-m on` 重新检测该二进制文件。
`-o` *output-binary-file*	将检测过的二进制文件写入指定的文件。缺省情况下，使用检测过的文件覆盖输入二进制文件。

如果对某个已检测的输入二进制文件运行 `uncover` 命令，`uncover` 将发出错误消息，指出已检测该二进制文件，无法再次检测，您可以运行该二进制文件来生成覆盖数据。

运行检测过的二进制文件

检测二进制文件后，您可以按正常方式运行它。每次运行检测过的二进制文件时，都会在 uncover 执行检测期间创建的、后缀为 .uc 的覆盖数据目录中收集代码覆盖数据。由于 uncover 数据收集是多线程安全的，并且是多进程安全的，因此，对进程中的并发运行数量或线程数量没有限制。覆盖数据将累积所有的运行和线程。

生成并查看覆盖报告

要生成覆盖报告，请对覆盖数据目录运行 uncover 命令。例如：

```
$ uncover a.out.uc
```

此命令将根据 a.out.uc 目录中的覆盖数据生成一个名为 *binary-name*.er 的 Oracle Solaris Studio 性能分析器实验目录，启动性能分析器 GUI，并显示该实验。当前目录或起始目录中存在 .er.rc 文件可能会影响性能分析器显示实验的方式。有关 .er.rc 文件的更多信息，请参见《Oracle Solaris Studio 12.4：性能分析器 》

可以生成 HTML 格式的报告在 Web 浏览器中查看它，或者生成 ASCII 格式的报告在终端窗口中进行查看。还可以将数据定向到代码分析器可以分析和显示它的目录。

-a	将错误数据写入 *binary-name*.analyze/coverage 目录以供代码分析器使用。
-c	启用指令、块和函数的执行计数报告。缺省情况下，仅报告已覆盖或未覆盖的代码的信息。（检测二进制文件和生成覆盖报告时，均指定该选项。）
-e on | off	确定是否为覆盖报告生成实验目录以及是否在性能分析器 GUI 中显示实验。缺省值为 on。
-H *html-directory*	在指定的目录中以 HTML 格式保存覆盖数据，并在 Web 浏览器中自动显示这些数据。
-h 或 -?	显示帮助。
-n	生成覆盖报告，但不启动性能分析器或 Web 浏览器等查看器。
-t *ascii-file*	在指定的文件中生成 ASCII 覆盖报告。
-V	输出 uncover 版本并退出。

-v 详细。输出 Uncover 正在执行的操作的日志。

仅启用一种输出格式。如果指定多个输出选项，则 uncover 使用命令中的最后一个选项。

例 2 uncover 命令示例

```
$ uncover a.out
```

此命令将检测二进制文件 a.out，覆盖输入 a.out，在当前目录中创建 a.out.uc 覆盖数据目录，并在 a.out.uc 目录中保存输入 a.out 的副本。如果已检测 a.out，将显示警告消息，并且不执行检测。

```
$ uncover -d coverage a.out
```

此命令在目录 coverage 中创建 a.out.uc 覆盖目录。

```
$ uncover a.out.uc
```

此命令会使用 a.out.uc 覆盖目录中的数据在工作目录中创建代码覆盖实验 (a.out.er)，并启动性能分析器以显示该实验。

```
$ uncover -H a.out.html a.out.uc
```

此命令使用 a.out.uc 覆盖目录中的数据在 a.out.html 目录中创建 HTML 代码覆盖报告，并在 Web 浏览器中显示该报告。

```
$ uncover -t a.out.txt a.out.uc
```

此命令使用 a.out.uc 覆盖目录中的数据在 a.out.txt 文件中创建 ASCII 代码覆盖报告。

```
$ uncover -a a.out.uc
```

此命令会使用 a.out.c 覆盖目录中的数据在 *binary-name*.analyze/coverage 目录中创建覆盖报告，以供代码分析器使用。

了解性能分析器中的覆盖报告

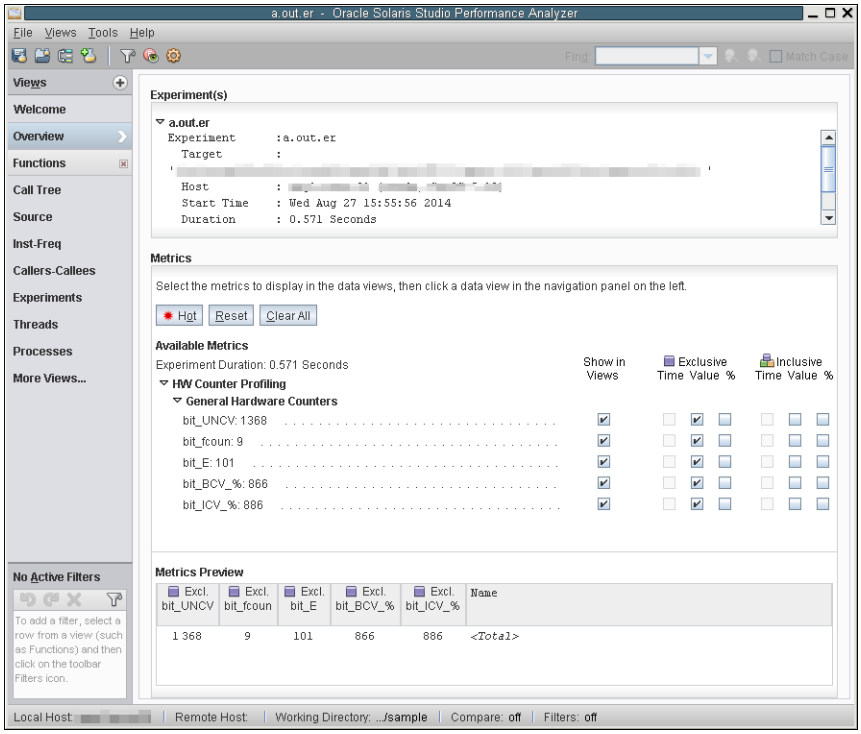
缺省情况下，在对覆盖目录运行 uncover 命令时，会在 Oracle Solaris Studio 性能分析器中以实验的形式打开覆盖报告。本节介绍显示覆盖数据的性能分析器界面。

有关性能分析器的更多信息，请参见集成帮助和《Oracle Solaris Studio 12.4：性能分析器》。

"Overview"（概述）屏幕

在性能分析器中打开覆盖报告时，将显示 "Overview"（概述）屏幕。此视图显示您正在运行的实验、实验的度量以及度量预览。

下图显示了性能分析器中的 "Overview"（概述）屏幕。



"Functions"（函数）视图

在导航面板中，单击 "Functions"（函数）视图可显示程序的函数和专用度量。要按照特定度量的值对数据进行排序，请单击所需的列标题。单击列标题下的箭头可以反转排序顺序。

这些度量包括以下各项：

`bit_UNCV` 未覆盖计数器，指示可为该函数覆盖的字节数。

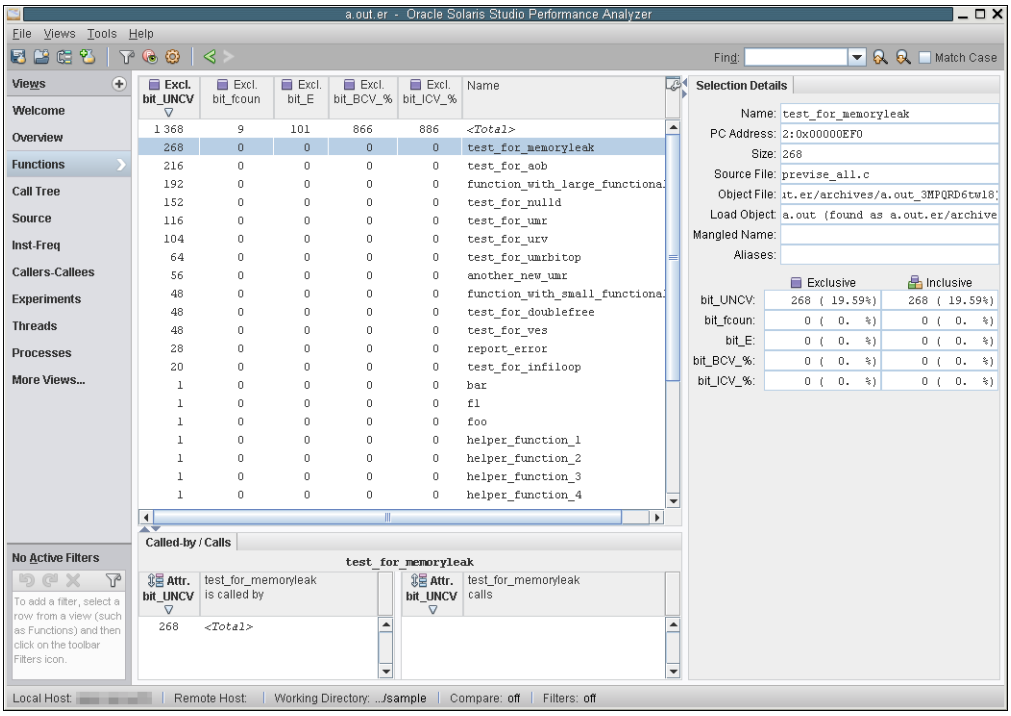
`bit_fcoun` 函数计数器，指示被覆盖的函数。

`bit_E` 指令执行计数器，指示是否在函数中执行了某个指令。

`bit_BCV_%` 块覆盖率计数器，指示在函数中覆盖的块所占的百分比。

`bit_ICV_%` 指令覆盖率计数器，指示在函数中覆盖的指令所占的百分比。

下图显示了性能分析器中的一个覆盖报告，按 `bit_UNCV` 排序。



"Uncoverage"（未覆盖）计数器 (`bit_UNCV`)

`bit_UNCV` 度量（也称为未覆盖计数器）是 `uncover` 的一项非常强大的功能。如果使用此列作为排序键，则在降序排列时，显示在最上面的函数是最有可能提高覆盖率的函数。在上图中，`test_for_memory_leak（）` 函数位于列表的顶端，因为它在 `bit_UNCV` 列中的数值最大。`function_with_small_functionality（）`、`test_for_doublefree（）` 和 `test_for_ves（）` 函数按字母表顺序列出，因为它们的数值相同。

`test_for_memory_leak（）` 函数的 `bit_UNCV` 数目是指，在向套件中添加一个测试以调用该函数时可能覆盖的代码字节数。根据函数的结构，覆盖率实际增加的量会有所不同。

如果该函数没有分支，并且它调用的所有函数也是直线型函数，则覆盖将增加指定的字节数。但是，覆盖增长通常会小于潜在值，也许会小很多。

bit_UNCV 列中使用非零值的未覆盖函数称为根未覆盖函数，表示它们都由覆盖函数调用。仅由非根未覆盖函数调用的函数没有自己的未覆盖数目。可以推定，在后续运行中，随着测试套件的改进而覆盖了潜力较大的未覆盖函数，这些函数将成为覆盖或未覆盖函数。

覆盖数目是非独占性的。

"Function Count"（函数计数）计数器 (bit_fcoun)

bit_fcoun 会报告已覆盖的函数和未覆盖的函数。如果计数为零，表示该函数未覆盖。如果计数非零，表示该函数已覆盖。如果执行了函数中的任一指令，该函数将视为已覆盖。

可以检测到此列中的非顶层未覆盖函数。如果某个函数的 bit_fcoun 为零，并且 bit_UNCV 也为零，则该函数不是顶层覆盖函数。

"Instr Exec"（指令执行）计数器 (bit_E)

bit_E 计数器会显示已覆盖的指令和未覆盖的指令。零计数表示未执行指令，非零计数表示已执行指令。

在 "Functions"（函数）视图中，该计数器会显示为每个函数执行的指令总数。此计数器还出现在 "Source"（源）视图和 "Disassembly"（反汇编）视图中。

"Block Covered %"（块覆盖率）计数器 (bit_BCV_%)

对于每个函数，"Block Covered %"（块覆盖率）计数器为 bit_BCV_%，该计数器显示该函数中已覆盖的基本块的百分比。此数值指示函数被覆盖的程度。请忽略 "Total"（总计）行中的此条目，它是列中百分比之和，没有意义。

"Instr Covered %"（指令覆盖率）计数器 (bit_ICV_%)

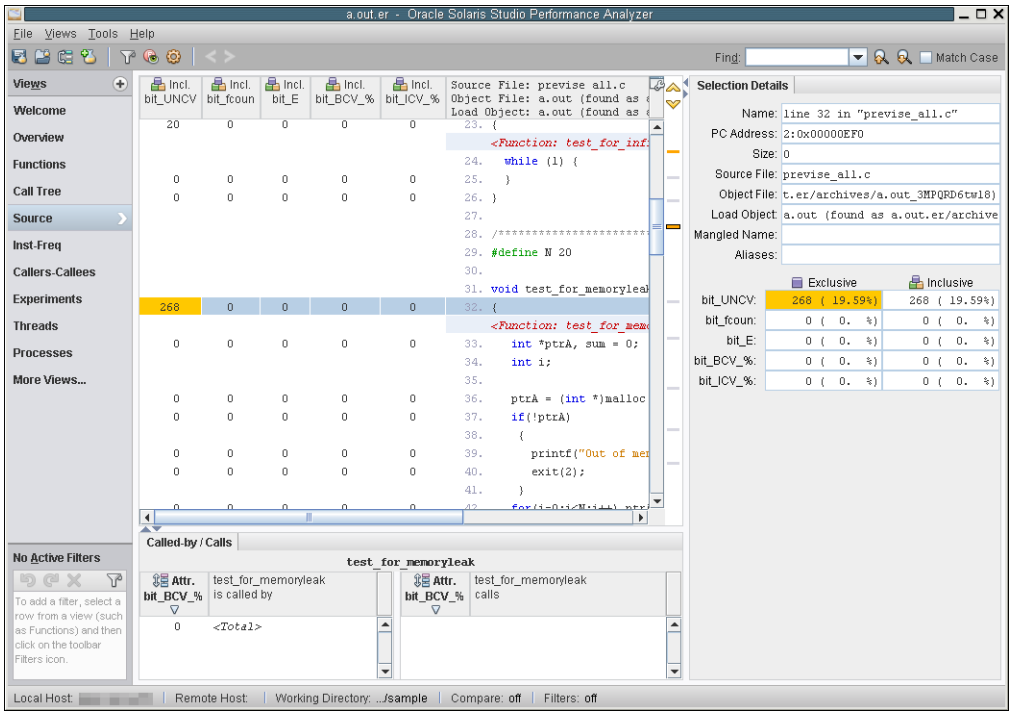
对于每个函数，bit_ICV_% 计数器会显示该函数中已覆盖指令百分比。此数值指示函数被覆盖的程度。请忽略 "Total"（总计）行中的此条目，它是列中百分比之和，没有意义。

"Source"（源）视图

如果二进制文件是使用 -g 选项编译的，"Source"（源）视图将显示程序的源代码。由于 uncover 在二进制文件级别上检测您的程序，并且已使用优化设置编译程序，因此，此视图中的覆盖信息很难解释。

"Source"（源）视图中的 bit_E 计数器显示了为每个源代码行执行的指令数，它本质上是语句级别的代码覆盖信息。非零值表示该语句已覆盖；零值表示该语句未覆盖。变量声明和注释没有 bit_E 计数。

下图显示了打开的 "Source"（源）视图的一个示例。



对于没有与其关联的任何覆盖信息的源代码行，行是空白的，任何字段中都没有数值。出现这些空行的原因如下：

- 注释、空白行、声明和其他语言结构不包含可执行的代码。
- 由于以下某个原因，编译器优化删除了与这些行对应的代码：
  - 代码从不执行（死代码）。
  - 代码可以执行，但为冗余代码。

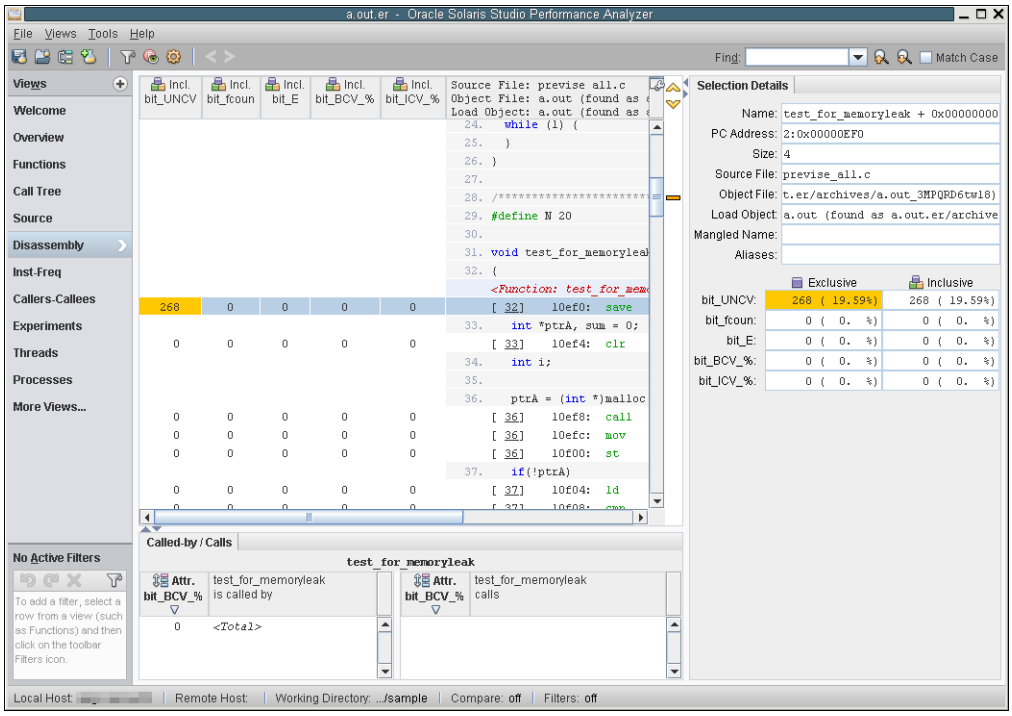
有关更多信息，请参见“使用 uncover 时的限制” [63]。

"Disassembly"（反汇编）视图

如果在 "Source"（源）视图中选择一行，然后选择 "Disassembly"（反汇编）视图，性能分析器将在二进制文件中查找选定的行，并显示其反汇编。

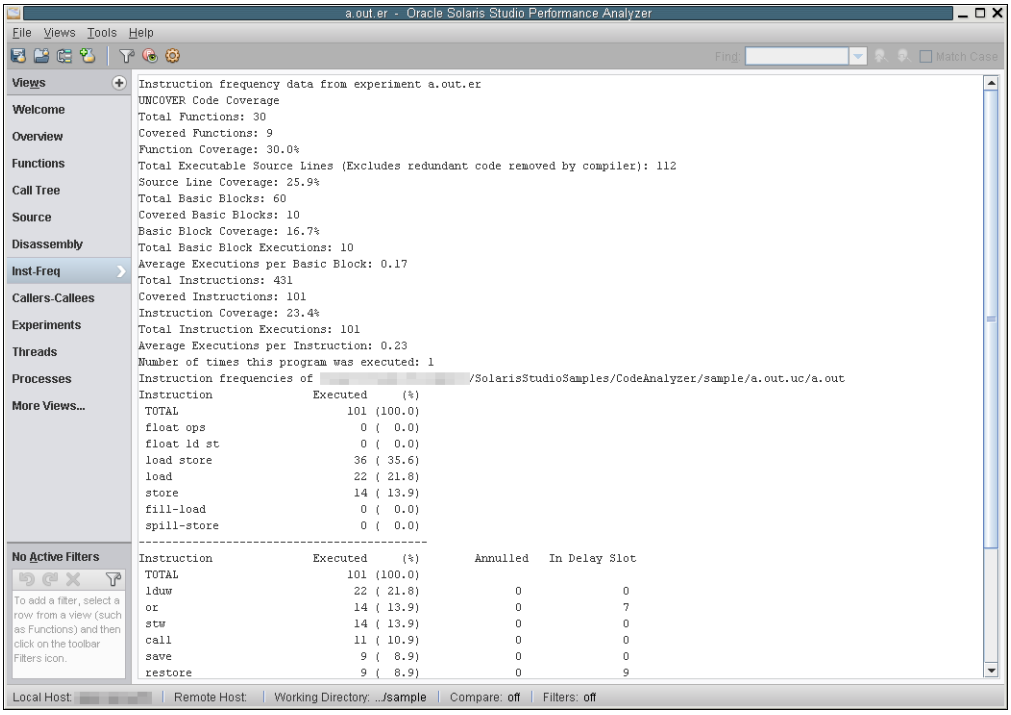
提示 - 如果未在 "View"（视图）窗格中看到 "Disassembly"（反汇编），请选择 "More Views..."（更多视图...），然后选中 "Disassembly"（反汇编）选项。

此视图中的 `bit_E` 计数器显示了每个指令的执行次数：



"Inst-Freq"（指令频率）视图

"Inst-Freq"（指令频率）视图会显示总体覆盖摘要。



了解 ASCII 覆盖报告

如果在从覆盖数据目录生成覆盖报告时指定了 `-t` 选项，`uncover` 会将覆盖报告写入指定的 ASCII 文件（文本文件）。

例 3 ASCII 覆盖报告样例

以下示例显示了一个 ASCII 覆盖报告样例。

```
UNCOVER Code Coverage
Total Functions: 95
Covered Functions: 58
Function Coverage: 61.1%
Total Basic Blocks: 568
Covered Basic Blocks: 258
Basic Block Coverage: 45.4%
Total Basic Block Executions: 564,812,760
Average Executions per Basic Block: 994,388.66
Total Instructions: 6,201
Covered Instructions: 3,006
Instruction Coverage: 48.5%
```

```
Total Instruction Executions: 4,760,934,518
Average Executions per Instruction: 767,768.83
Number of times this program was executed: unavailable
Functions sorted by metric: Exclusive Uncoverage

Excl.       Excl.     Excl.      Excl.       Name
Uncoverage  Function  Block      Instr
Count     Covered %  Covered %
13404       6004876   5464       5384        <Total>
1036             0      0          0        main
980              0      0          0        iofile
748              0      0          0        do_vforkexec
732              0      0          0        callso
708              0      0          0        do_forkexec
648              0      0          0        callsx
644              0      0          0        sigprof
644              0      0          0        sigprofh
556              0      0          0        do_chdir
548              0      0          0        correlate
492              0      0          0        do_popen
404              0      0          0        pagethrash
384              0      0          0        so_cputime
384              0      0          0        sx_cputime
348              0      0          0        itimer_realprof
336              0      0          0        ldso
304              0      0          0        hrv
300              0      0          0        do_system
300              0      0          0        do_burncpu
300              0      0          0        sx_burncpu
288              0      0          0        forkcopy
276              0      0          0        masksignals
256              0      0          0        sigprof_handler
256              0      0          0        sigprof_sigaction
216              0      0          0        do_exec
196              0      0          0        iotest
176              0      0          0        closeso
156              0      0          0        gethrustime
144              0      0          0        forkchild
144              0      0          0        gethrpxtime
136              0      0          0        whrlog
112              0      0          0        masksig
92               0      0          0        closesx
84               0      0          0        reapchildren
36               0      0          0        reapchild
32               0      0          0        doabort
8                0      0          0        csig_handler
0                1     66         72        acct_init
0                1    100        100        bounce
0               63    100         96        bounce_a
0               60    100        100        bounce-b
0               16     71         58        check_sigmask
0                1     83         77        commandline
0                1    100         98        cputime
0                1    100         98        dousleep
```

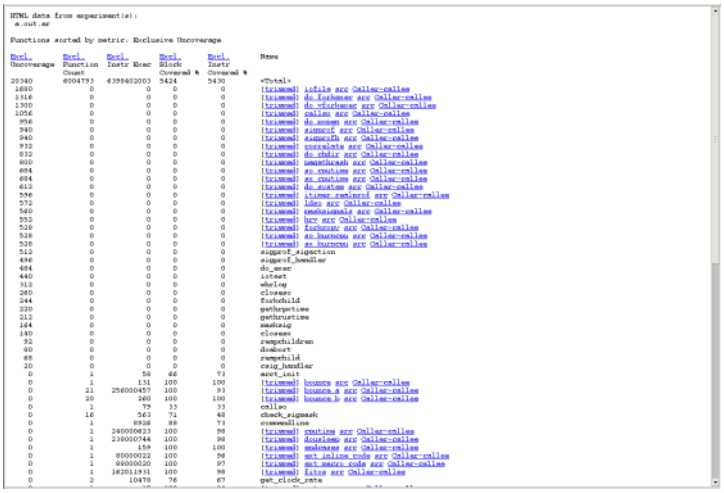
```
0             1    100        100        endcases
0             1    100         95        ext_inline_code
0             1    100         96        ext_macro_code
0             1    100         99        fitos
0             2     81         80        get_clock_rate
0             1    100        100        get_ncpus
0             1    100        100        gpf
0             1    100        100        gpf_a
0             1    100        100        gpf_b
0            10    100         93        gpf_work
0             1    100         97        icputime
0             1    100         96        inc_body
0             1    100         96        inc_brace
0             1    100         95        inc_entry
0             1    100         95        inc_exit
0             1    100         96        inc_func
0             1    100         94        inc_middle
0             1     57         72        init_micro_acct
0             1     50         43        initcksig
0             1    100         95        inline_code
0             1    100         95        macro_code
0             1    100         98        muldiv
0       6000000    100        100        my_irand
0             1    100         98        naptime
0            19     50         83        prdelta
0            21    100        100        prhrdelta
0            21    100        100        prhrvdelta
0             1    100        100        prtime
0           552    100         98        real_recurse
0             1    100        100        recurse
0             1    100        100        recursedeep
0             1    100         95        s_inline_code
0             1    100        100        sigtime
0             1    100         95        sigtime_handler
0            19    100        100        snaptod
0             1    100        100        so_init
0             2     66         75        stpwtch_alloc
0             1    100        100        stpwtch_calibrate
0             2     75         66        stpwtch_print
0          2002    100        100        stpwtch_start
0          2000     90         91        stpwtch_stop
0             1    100        100        sx_init
0             1    100         99        systime
0             3    100         95        tailcall_a
0             3    100         95        tailcall_b
0             3    100         95        tailcall_c
0             1    100        100        tailcallopt
0             1    100         97        underflow
0            21     75         71        whrvlog
0            19    100        100        wlog

Instruction frequency data from experiment a.out.er

Instruction frequencies of /export/home1/synprog/a.out.uc
```

```
Instruction              Executed     ()
TOTAL                  4760934518 (100.0)
float ops              2383657378 ( 50.1)
float ld st            1149983523 ( 24.2)
load store             1542440573 ( 32.4)
load                    882693735 ( 18.5)
store                   659746838 ( 13.9)
-----------------------------------------
Instruction              Executed     ()      Annulled   In Delay Slot
TOTAL                  4760934518 (100.0)
add                     713013787 ( 15.0)           16         1501335
subcc                   558774858 ( 11.7)            0            6002
br                      558769261 ( 11.7)            0               0
stf                     432500661 (  9.1)          726        36299281
ldf                     408226488 (  8.6)           40       103000396
faddd                   391230847 (  8.2)            0               0
fdtos                   366200726 (  7.7)            0               0
fstod                   360200000 (  7.6)            0               0
lddf                    288250336 (  6.1)          500       282200229
stw                     138028738 (  2.9)        26002        25974065
lduw                    118004305 (  2.5)           71        94000270
ldx                      68212446 (  1.4)            0            2000
stx                      68211370 (  1.4)            7        23532716
fitod                    36026002 (  0.8)            0               0
sethi                    36002986 (  0.8)            0             228
fdtoi                    30000001 (  0.6)            0               0
fdivd                    26000088 (  0.5)            0               0
call                     22250348 (  0.5)            0               0
srl                      21505246 (  0 5)            0              21
stdf                     21006038 (  0.4)            0               0
or                       19464766 (  0.4)            0        10981277
fmuls                     6004907 (  0.3)            0               0
jmpl                      6004853 (  0.1)            0               0
save                      6004852 (  0.1)            0               0
restore                   6002294 (  0.1)            0         6004852
sub                       6000019 (  0.1)            0               0
xor                       6000000 (  0.1)            0               0
fitos                     6000000 (  0.1)            0               0
fstoi                     6000000 (  0.1)            0               0
and                       6000000 (  0.1)            0               0
andn                      6000000 (  0.1)            0               0
sll                       3505225 (  0.1)            0               0
nop                       3505219 (  0.1)            0         3505219
fxtod                        7763 (  0.0)            0               0
bpr                          6000 (  0.0)            0               0
fcmped                       4837 (  0.0)            0               0
fbr                          4837 (  0.0)            0               0
fmuld                        2850 (  0.0)            0               0
orcc                          383 (  0.0)            0               0
sra                           241 (  0.0)            0               0
ldsb                          160 (  0.0)            0               0
mulx                           87 (  0.0)            0               0
stb                            31 (  0.0)            0               0
mov                            21 (  0.0)            0               0
```

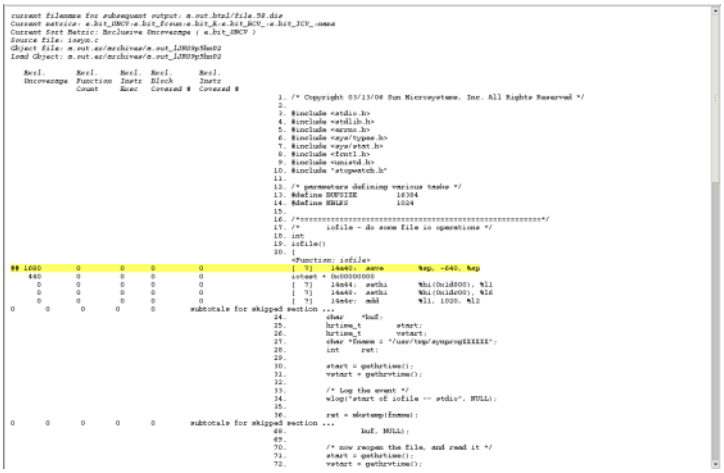
```
fdtox                                15 (  0.0)            0               0
============================================================
```

了解 HTML 覆盖报告

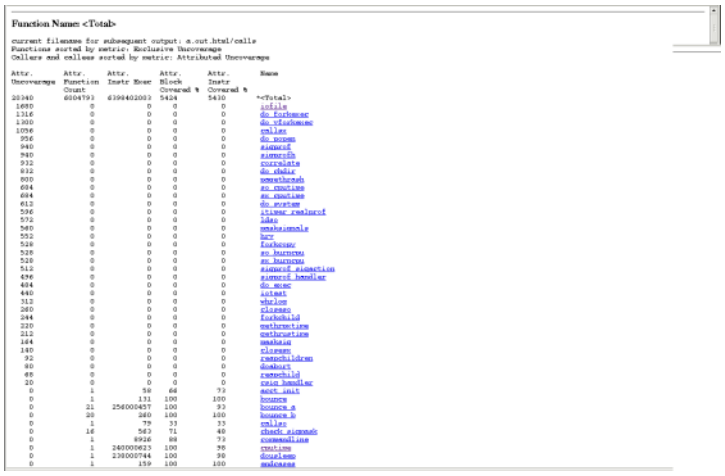
HTML 报告类似于性能分析器中显示的报告：



单击某个函数的函数名称链接或 `trimmed` 链接，将显示该函数的反汇编数据：



单击函数的 Caller-callee 链接可显示调用方-被调用方数据：



使用 uncover 时的限制

本节介绍在使用 uncover 时的已知限制。

只能检测有注释的代码

uncover 实用程序只能检测按照“使用 uncover 的要求” [49]中的说明准备的代码。无注释代码可能来自链接到二进制文件的汇编语言代码，或者来自使用早于本节所列版本的编译器或操作系统编译的模块。

uncover 无法检测汇编语言模块或包含 asm 语句或 .il 模板的函数。

编译器选项影响生成的代码

uncover 与使用以下任一编译器选项生成的二进制文件不兼容：

- -p
- -pg
- -qp

- -xpg
- -xlinkopt

计算机指令可能不同于源代码

uncover 实用程序可处理计算机代码。它会查找计算机指令的覆盖，然后将此覆盖与源代码相关联。某些源代码语句没有关联的计算机指令，因此，uncover 看上去似乎并未报告这些语句的覆盖。

例 4 简单示例

考虑以下代码片段：

```
#define A 100
#define B 200
...
if (A>B) {
...
}
```

您可能希望使用 uncover 报告 if 语句的非零执行计数。但是，编译器可能会删除此代码。uncover 不会在检测过程中检测到它，因此不会报告这些指令的覆盖信息。

例 5 死代码示例

以下示例显示了死代码：

```
1  void foo()
2  {
3     A();
4     return;
5     B();
6     C();
7     D();
8     return;
9   }
```

对应的汇编显示删除了 B,C,D 的调用，因为该代码从不执行。

```
foo:
.L900000109:
/* 000000          2 */         save    %sp,-96,%sp
/* 0x0004          3 */         call    A       ! params =      ! Result =
/* 0x0008            */         nop
/* 0x000c          8 */         ret     ! Result =
/* 0x0010            */         restore %g0,%g0,%g0
```

因此，不会针对第 5 行到第 6 行报告覆盖。

```
   Excl.       Excl.     Excl.  Excl.      Excl.
Uncoverage  Function  Instr  Block      Instr
Count     Exec   Covered %  Covered %
1.  void foo()
## 0           1         1      100        100              2.   {
<Function: foo
## 0           0         2        0          0              3.      A();
4.      return;
5.      B();
6.      C();
7.      D();
8.     return;
## 0           0         2        0          0              9.    }
```

例 6　　冗余代码示例

以下示例显示了冗余代码：

```
      1  int g;
2  int foo() {
3    int x;
4    x = g;
5    for (int i=0; i<100; i++)
6        x++;
7    return x;
8  }
```

在低优化级别下，编译器可能会为所有行生成代码：

```
foo:
.L900000107:
/* 000000          3 */         save    %sp,-112,%sp
/* 0x0004          5 */         sethi   %hi(g),%l1
/* 0x0008            */         ld      [%l1+%lo(g)],%l3 ! volatile
/* 0x000c            */         add     %l1,%lo(g),%l2
/* 0x0010          6 */         st      %g0,[%fp-12]
/* 0x0014          5 */         st      %l3,[%fp-8]
/* 0x0018          6 */         ld      [%fp-12],%l4
/* 0x001c            */         cmp     %l4,100
/* 0x0020            */         bge,a,pn        %icc,.L900000105
/* 0x0024          8 */         ld      [%fp-8],%l1
.L17:
/* 0x0028          7 */         ld      [%fp-8],%l1
.L900000104:
/* 0x002c          6 */         ld      [%fp-12],%l3
/* 0x0030          7 */         add     %l1,1,%l2
/* 0x0034            */         st      %l2,[%fp-8]
/* 0x0038          6 */         add     %l3,1,%l4
/* 0x003c            */         st      %l4,[%fp-12]
/* 0x0040            */         ld      [%fp-12],%l5
/* 0x0044            */         cmp     %l5,100
/* 0x0048            */         bl,a,pn %icc,.L900000104
/* 0x004c          7 */         ld      [%fp-8],%l1
/* 0x0050          8 */         ld      [%fp-8],%l1
```

```
.L900000105:
/* 0x0054          8 */         st      %l1,[%fp-4]
/* 0x0058            */         ld      [%fp-4],%i0
/* 0x005c            */         ret     ! Result =  %i0
/* 0x0060            */         restore %g0,%g0,%g0
```

在高优化级别时，大多数可执行的源代码行不具有任何对应的指令：

```
foo:
/* 000000          5 */         sethi   %hi(g),%o5
/* 0x0004            */         ld      [%o5+%lo(g)],%o4
/* 0x0008          8 */         retl    ! Result =  %o0
/* 0x000c          5 */         add     %o4,100,%o0
```

因此，不会针对某些行报告覆盖。

```
     Excl.       Excl.     Excl.  Excl.      Excl.
Uncoverage  Function  Instr  Block      Instr
Count     Exec   Covered %  Covered %
1. int g;
0           0         0        0          0               2. int foo() {
<Function foo>
3.   int x;
4.   x = g;

Source loop below has tag L1
Induction variable substitution performed on L1
L1 deleted as dead code
## 0           1         3      100        100               5.   for (int i=0; i<100; i++)
6.       x++;
7.   return x;
0           0         1        0          0               8. }
```

索引

E

G

J

S

U

W

X

Y