DASSAULT SYSTEMES

使用ENOVIA Synchronicity DesignSync Data Manager 创建多站点知识产权块发布的分布式系统

目录

摘要

半导体市场中竞争优势与成本控制的一个关键要素是对复杂生产数据的高效管理。通过以小规模并可重复使用的数据集合方式管理，此类数据（或称之为知识产权）可以为不同项目的多个团队共享使用，确保质量控制并消除重复工作。该白皮书介绍了如何定制ENOVIA Synchronicity DesignSync Data Manager (DesignSync)，使其通过高效分布处理提供集中存储、版本控制与资源管理，成为管理知识产权的分布式系统。

简介

设计团队使用DesignSync来管理不同设计中心间可重复使用的知识产权数据块（称为IP块）。在产品开发的过程中，所有团队开发出数千个IP块和块的配置方法。每个块经设计、测试确保稳定后，将被发布用于企业内部其他设计团队使用。这里有几种方法可以帮助企业优化DesignSync，从而在企业内部多站点复杂网络间快速高效地管理IP块。该白皮书使用了DesignSync的部分组件概述了这些方法：模块、镜像、触发器、反向引用，文件缓存及模块缓存。

该解决方案文档面向负责确保最优化重复使用企业IP资产的设计研究人员、项目领导人及IP开发团队。在DesignSync V6R2011版中拥有此项功能。“自建镜像”的选项仅在V6R2012版中可用。

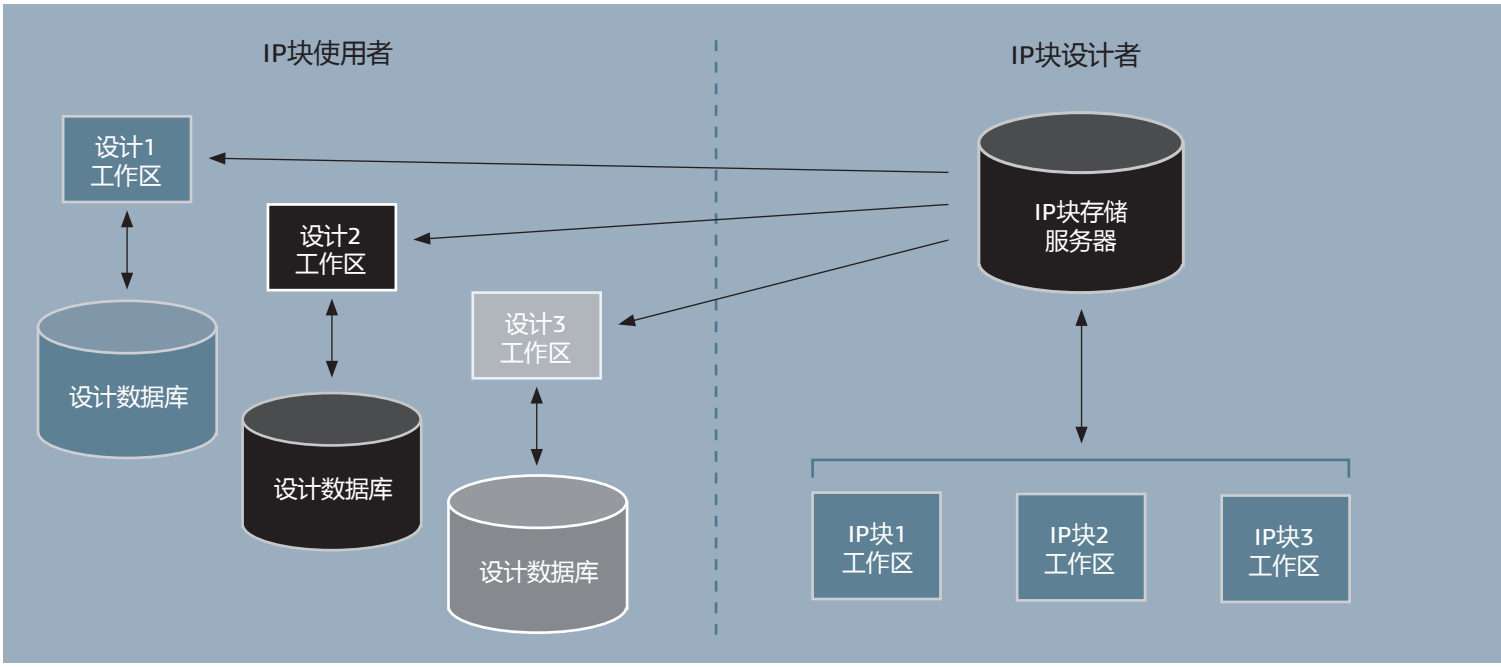

图 1

首选解决方案目标

首选解决方案的目标是：

● 各设计中心可轻松获取相关IP块最新发布的版本。IP块进行更新后会自动将新的发布结果传送至使用这些IP块的设计中心。

● 花费最小开支部署多站点IP块发布的分布式系统

● 允许各设计中心选择在其站点上发布的IP块。

● 实现网络与服务器的高效运行，确保大量块的发布不影响正常业务的操作。

● 允许创建和分发由其他块继承而来的块。通常需要提供继承块的报告。见图2。

附加扩展的目标

首选解决方案之外可用扩展的目标是：

● 允许获取某个IP块发布的多个版本来支持多个正在进行的设计。

● 当新版本发布时自动更新块并删除已不再用于任何设计的旧版本。

● 通过清理在局域网内相同块的多个冗余副本确保在不同版本间统一数据文件的最优重复使用。

● 当新块发布时允许设计中心的特定人员得到通知并提供在其指定选择下推送块的高效方式。

● 允许设计中心访问正在开发的IP块的"开发中"版本。

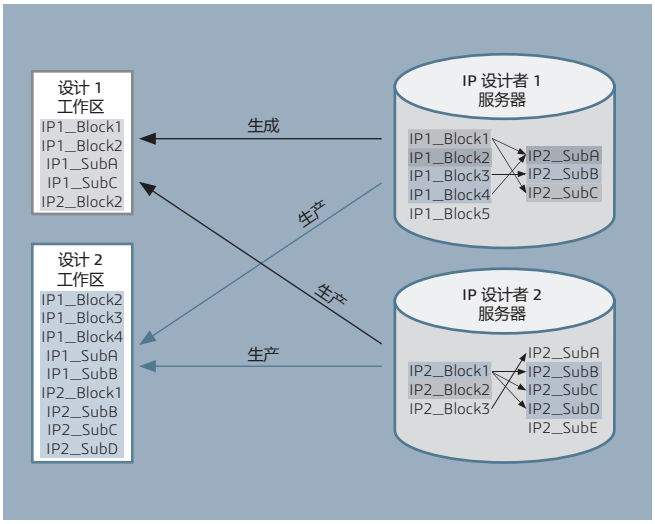

图 2

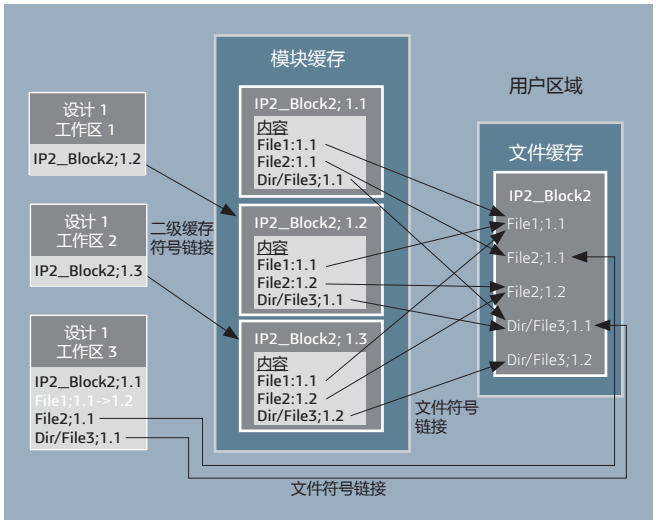

图 3

首选方案概述

使用模块管理知识产权块

管理知识产权每个块的数据的理想方式是使用模块。块的数据是受独立修订控制的，且一系列数据集合又接受模块不同版本的修订控制。随着文件生成不同独立的版本，模块的版本也会出现不同的分支，以便在管理组成块的文件集合时提供有效的控制。

另外，模块还有个好处就是允许组成块数据的目录结构出现变化。对模块旧版本的获取将重建模块旧版本创建时定义的目录结构。该"目录版本控制"的特性只限于针对模块而不是DesignSync对文件型数据的管理。

最终，模块考虑到了块分级引用并使用任何数量子模块的情况。在配置分级引用连接并引用其子块的时候必须预先注意，以防止在同一目录下获取到相同块的不同版本。若要了解更多详细信息，请参考章节"创建顶层容器模块用于识别各设计中心需要使用的块"。

该白皮书提示假设阅读者已熟悉DesignSync的模块功能。该整体功能被称之为"模块组"，而其管理的单一数据集合被称之为"模块"。

镜像解决方案

使用镜像更新在远程设计中心的模块

镜像"推送"系统是使用特定选择器将块推送给设计中心的一种有效方式。如果选择器是动态的（如分支标签或模块版本标签），则无论何时若该动态选择器变化（由于新模块版本正在进入分支或该标签正在转移到新模块的版本中），该镜像系统将自动获取该模块的数据并推送给设计中心。

使用镜像系统的一项挑战是随着被推送块的增多，检阅镜像系统所需的时间也会增加。针对每个新增块，设计中心的管理员必须在其站点创建新的镜像指向其块的模块URL。当块已不再需要使用时，管理员必须从镜像中删除其相关的定义及模块数据。这对管理员根据日常状态报告的监视、分析及管理都将带来挑战。

每个被定义的镜像在托管知识产权块的DesignSync集中服务器上都有一个相应的入口。由于必须持续监视每个块的版本状态，服务器必须有能力承受由各设计中心创建的大量镜像带来的处理负荷。

以下概要介绍的解决方案可克服这些障碍。

解决方案详细介绍

我们用托管在SyncServer"iphost:80"上的块样本集合作为例子。某些块的子集会被分发并更新到两个设计中心。图4展示了通过DesignSync GUI模块层次结构浏览器（通过模块显示模块层级调用）看到的该服务器上所有模块的"showmodes"列表和两个容器模块（及其相应的层次）的列表。

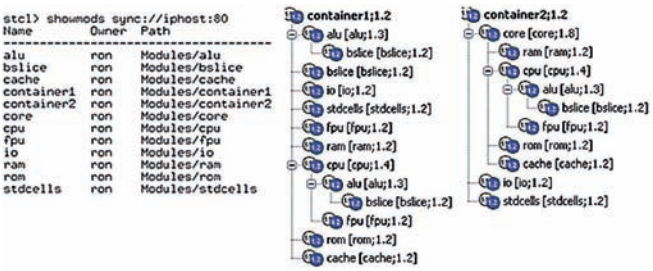

图 4

所有块/模块使用名为RELEASED的标签来标记对设计中心可用的模块版本。在该简化的配置过程中，每个模块仅存在一个RELEASED的模块版本。在"跟踪每个块的多个版本"章节中将讨论多个已发布模块更复杂的配置。

部分块由子模块层级定义。图5展示了组成"cpu;RELEASED"模块版本的详细模块层级清单。core使用ram，rom，cache及cpu创建，而cpu用到alu和fpu，且最终alu用到bslice。从任何父级模块到其子模块之一的分级引用使用RELEASED标签作为选择器。

```
stcl> showhrefs -rec -report brief [url vault core]\:RELEASED
Name                Url
-------------------------------------------------------------------
core:RELEASED       sync://iphost:80/Modules/core:RELEASED
    ram             sync://iphost:80/Modules/ram:RELEASED
    cpu             sync://iphost:80/Modules/cpu:RELEASED
        alu         sync://iphost:80/Modules/alu:1.3
            bslice  sync://iphost:80/Modules/bslice:1.2
        fpu         sync://iphost:80/Modules/fpu:1.2
    rom             sync://iphost:80/Modules/rom:RELEASED
    cache           sync://iphost:80/Modules/cache:RELEASED
```

图 5

当hrefmode命令置于"正常"（命令的默认配置，无需在命令行中指定）时，"showhrefs"命令可以遍历模块的层级。当命令解除引用RELEASED标签时，它可以找到"core"模块带标签的模块版本，在core;RELEASED中获得分级引用的列表——在该例子中为ram;RELEASED, cpu;RELEASED, rom;RELEASED和cache;RELEASED。在"showhrefs"命令遍历到子模块上时其依然处于正常hrefmode。当"populate"命令通过某个带标签的模块版本访问子模块版本时，该命令将自动切换到静态hrefmode。从这里往前的每个子模块都可以被存储在href的静态模块版本找到。cpu;RELEASED是通过版本标签访问到的一个子模块，因此，当"populate"在cpu;RELEASED中跟随hrefs时，它会切换到静态hrefmode。在cpu;RELEASED中使用的alu的静态版本是1.3。fpu的静态版本是1.2。在alu;1.3中使用的bslice的静态版本是bslice;1.2。

请注意如果一个父模块的分级引用指向子模块的分支标签，则当"populate"命令访问该子模块时，不会从正常切换到静态hrefmode——它将一直保持在正常模式。这是由模块版本使用分支标签的动态方式决定的。在新模块版本进入分支时，它可以随时变化。如果父模块的分级引用指向子模块的唯一版本ID（如href指向"alu;1.18.1.4"），则在该子模块内处理任何模块分级时，"populate"将切换到静态hrefmode。

重要提示：hrefmodes的切换是在处理模块分级时必须学习并理解的重要概念。

创建顶层容器模块用于识别各设计中心需要使用的块

一个“容器”模块很可能并没有与其关联的特定数据文件。它只是定义了指向某个设计中心正在使用并需要自动更新的每个块的分级引用。

在范例数据中有两个容器模块：“container1”定义了在设计中心1中需要保持更新的已发布模块的集合；“container2”定义了在设计中心2中需要保持更新的已发布模块的集合。

“container1”有9个被定义的分级引用：分别指向alu, bslice, io, stdcells, fpu, ram, cpu, rom和cache的RELEASED模块版本。

“container2”只有3个被定义的分级引用：分别指向core，io和stdcells的RELEASED模块版本。

分级引用可以通过“addhref”命令创建。DesignSync将为模块添加多个分级引用并创建单一模块版本变得非常简单。（在之前的版本中，模块版本每新增一次都可能带来一个新的模块版本。）创建多个分级引用最简单的方式是定义一个每行包含每个待添加分级引用相关信息的文件。以下是用于创建“container1”中分级引用的文件和“addhref”命令的调用。

```
stcl> exec cat cont1_hreffile
url sync://iphost:80/Modules/stdcells selector RELEASED
url sync://iphost:80/Modules/io selector RELEASED
url sync://iphost:80/Modules/cache selector RELEASED
url sync://iphost:80/Modules/ram selector RELEASED
url sync://iphost:80/Modules/rom selector RELEASED
url sync://iphost:80/Modules/cpu selector RELEASED
url sync://iphost:80/Modules/fpu selector RELEASED
url sync://iphost:80/Modules/alu selector RELEASED
url sync://iphost:80/Modules/bslice selector RELEASED

stcl> addhref container1 -rootpath .. -file cont1_hreffile
```

图 6

当自动定义子模块基本目录与父模块基本目录相关的相对路径时，“addhref”命令提供了一个“-rootpath”选项用来定义应加在模块名称之前的路径。

在图7中定义了根目录“..”。因此，子模块基本目录的相对路径是../stdcells,../io及../cache。这就使得“container1”和所有子模块的基本目录在其父目录中互相对等。使用“..”相对路径是表示模块层级的标准使用方式。“-rootpath”选项只是可以更容易指定这些相对路径。

以下是在递归获取“container1”和“container2”后的列表：

```
stcl> ls center1_Blocks
Time Stamp           WS Status  Version   Type       Name
----------           ---------  -------   ----       ----
01/22/2011 13:29                                    alu
01/22/2011 13:29                                    bslice
01/22/2011 13:29                                    cache
01/22/2011 13:29                                    container1
01/22/2011 13:29                                    cpu
01/22/2011 13:29                                    fpu
01/22/2011 13:29                                    io
01/22/2011 13:29                                    ram
01/22/2011 13:29                                    rom
01/22/2011 13:29                                    stdcells

stcl> ls center2_Blocks
Time Stamp           WS Status  Version   Type       Name
----------           ---------  -------   ----       ----
01/22/2011 13:30                                    container2
01/22/2011 13:37                                    core
01/22/2011 13:30                                    io
01/22/2011 13:30                                    stdcells
```

图 7

仅有容器模块的分级引用使用“../<moduleName>”这样的相对路径。任何包含模块层级的IP块（如“core”和“cpu”）使用默认“./<moduleName>”相对路径（相对父模块目录）。这是因为在这些IP块中使用的子模块都是IP块其自身（如“alu”在“cpu”中使用，但其自身也是一个已发布、分发并重复使用的IP块）。若在任何地方过多使用“..”相对路径，则很可能导致“populate”尝试在相同的基本目录位置以不同的选择器获取相同的子模块——这是一个不允许的错误。若必须在任何地方使用“..”相对路径，则所有获取的模块都将拥有对等的基本目录，因此每个放到工作区中的模块实例都可能需要在分级引用相对路径属性中的唯一基本目录名称。

定义镜像为每个设计中心获取容器模块

可以创建一个容器模块定义一系列块来确保在每个设计中心对块的获取和更新。“mirror create”命令可以用于创建一个镜像递归推送并获取容器。也可以通过在SyncServer控制面板中的“Add Mirror”功能实现。见图8。

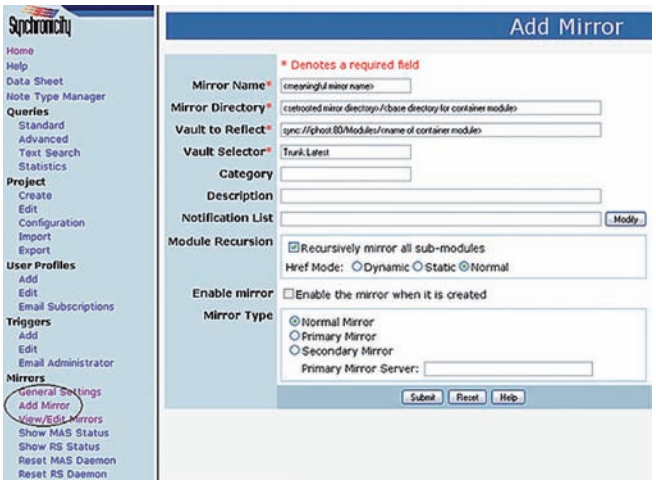

图 8

新创建的镜像尚未被标记为启用。在启用镜像之前，用户可以按照以下步骤完成配置。

“镜像目录”域并不包含镜像自身的基本目录。相反，由于镜像更新过程（MUP）运行了“populate-recursive”，它被设置为存放正在获取的容器模块的基本目录，而连带其“../<moduleName>”相对路径获取到的子模块将为容器基本目录侧的模块创建基本目录。在单一的镜像空间内需要所有这些模块，特别是该镜像将用作模块缓存或mcache时。

使用分享模式自动获取模块数据文件到文件缓存中

在启用镜像并在镜像中更新模块前，最好对镜像管理服务器（MAS）进行定制，将“-share”选项添加到“populate”中。“-share”选项可使所有模块成员的文件提取到DesignSync文件缓存中（缓存位置在DesignSync客户端应用安装完成后确定）。镜像中的模块成员就可以创建符号链接以指向到缓存中的文件。

注意：将模块成员文件放到缓存中可以优化系统性能，这是因为：

- 默认情况下，在搜索特定文件的版本时，“populate”命令使用其“-fromlocal”选项开始浏览文件缓存。若发现文件的所需版本，它将被从缓存而不是服务器中取出。这将最大程度降低网络和服务器的负载，减少用户响应时间。
- 用户可以使用“populate-share”来获取整个模块（仅针对Unix）。这将在用户工作区域中创建模块的目录结构并创建到文件缓存中每个模块成员的符号链接。该过程减少了重复文件的数量。

● 当发布分布式系统扩展到允许块的多个版本在同一个设计中心进行镜像时，一个模块版本实例中的成员文件很可能将在不同版本间保持静态。通过在"-share"模式中获取所有模块版本，每个模块版本的文件都将存放到文件缓存中。一旦所需文件版本存在缓存中，任何使用该文件版本的其他模块版本都将仅获得指向该文件版本的符号链接（或可选硬链接）。这样可以在文件系统上腾出更多的空间，因为相同文件的不同副本不再以不同模块版本形式存在。请参考图3中图形化说明了解符号链接是如何为受控文件保存单一本地副本的。

若要调用MAS将"-share"选项添加到"populate"命令中，请在MAS注册表配置中更新PortRegistery.reg文件：

HKEY_LOCAL_MACHINE\Software\
Synchronicity\General\Mirrors\MUP\
PopulateOptions

该键值的默认值是"-force –get –retain"，用户可以添加额外命令选项如"-share"。

通过"populate"创建硬链接取代动态符号链接（软链接）可以是文件系统资源的使用变得更加高效。后续章节将作为基本方法的扩展讨论该方法。

性能优化的注册表配置

"Add Mirror"控制面板可用于使用"Trunk:Latest"选择器递归更新容器模块。因此，容器模块的新版本得到进入（很可能带有涉及将要更新到设计中心的IP块的分级引用变更）。这些容器模块的新版本都将自动取出到镜像目录中。

通常，当镜像执行递归"populate"时，每个取出的子模块都将自动定义其为自身的镜像（称之为子镜像）。在范例中，总共仅有10个模块可以获得，且仅定义了10个子镜像。当有500个块可以被取出，有5个设计中心拥有其自己的容器和镜像配置来依次获取500个块，这将可能导致突然有2500个子镜像被定义。部分客户甚至需要超过35000个特定的镜像。

为避免这个问题，我们允许客户端MUP通过修改注册表禁用递归获取模块时自动创建子镜像。请在镜像管理服务器（MAS）注册表配置中更新PortRegistery.reg文件：

HKEY_LOCAL_MACHINE\Software\
Synchronicity\General\Mirrors\Options\
ModuleRegisterSubmirrors=dword:0

因为在注册表键值设置为"dword:0"时不会创建子镜像，这就意味着对于5个设计中心500个块的范例来说，将总共仅创建5个镜像。这将大幅减少SyncServer托管的IP块的成本，系统不必再针对定义的镜像集合执行每个版本控制操作。

另一个避免"populate"命令经常在MAS镜像系统中运行的方法是为你的模块镜像配置注册表添加"-incremental"选项来优化性能：

HKEY_LOCAL_MACHINE\Software\
Synchronicity\General\Mirrors\Options\
ModuleIncrPopOnly=dword:1

所有PortRegistery.reg自定义配置完成后，用户就可以通过"mirror enable"命令或View/Edit Mirrors控制面板来启用镜像了。

扩展镜像系统使得当IP块变化时更新容器

系统的目标之一是每次当引用的容器子模块发布时，镜像就可以获取到容器模块。（如子模块的“RELEASE”标签移动到该模块的新版本）。简单地定义镜像及禁用自动创建镜像还不够，因为当块发生变化且其RELEASED标签移动到该块的新模块版本中时，这将并不能导致容器模块变化。由于自动创建子镜像已被关闭，在SyncServer上已不存在指示设计中心需要更新其相关容器的配置。解决这种情况方案可以使用DesignSync的其他两项功能：“where-used”反向引用和服务端触发器。

“Whereused”反向引用

每当运行“addhref”命令创建从父模块到子模块的分级引用时，系统将存储一个“反向引用”。该反向引用表明哪个父模块包含到子模块的分级引用。“whereused”命令可以分析反向引用的指针并报告父模块的关联。图9展示了针对“bslice;RELEASED”块递归的“whereused”输出。（通过ModulesShowWhere Used menu item，同样可以用DesignSync GUI使用该输出的图形化版本）。详见图9。

```
stcl> whereused -recursive -versions RELEASED -showtags all sync://iphost:80/Modules/bslice

sync://iphost:80/Modules/bslice:1.2 - Latest, RELEASED, Trunk:
   sync://iphost:80/Modules/alu:1.3 - Latest, RELEASED, Trunk:
       sync://iphost:80/Modules/cpu:1.3 - Trunk:
       sync://iphost:80/Modules/cpu:1.4 - Latest, RELEASED, Trunk:
           sync://iphost:80/Modules/core:1.3 - Trunk:
           sync://iphost:80/Modules/core:1.4 - Trunk:
           sync://iphost:80/Modules/core:1.5 - Trunk:
           sync://iphost:80/Modules/core:1.6 - Trunk:
               sync://iphost:80/Modules/container2:1.2 - Latest, Trunk:  <== 1
           sync://iphost:80/Modules/core:1.7 - Trunk:
           sync://iphost:80/Modules/core:1.8 - Latest, RELEASED, Trunk:
           sync://iphost:80/Modules/container1:1.2 - Latest, Trunk:      <== 2
       sync://iphost:80/Modules/container1:1.2 - Latest, Trunk:          <== 3
   sync://iphost:80/Modules/container1:1.2 - Latest, Trunk:              <== 4
```

图 9

由于反向引用的加入，每上一层级都将缩进。

标注“<==1”的行表示访问到容器2;1.2。从上文看，缩进的级别显示出容器2;1.2的引用core;1.6，引用cpu;1.4，引用alu;1.3，引用bslice;1.2（=bslice;RELEASED）。

标注“<==2”的行表示访问到容器1;1.2。从上文看，缩进的级别显示出容器1;1.2的引用core;1.6，引用cpu;1.4，引用alu;1.3，引用bslice;1.2（=bslice;RELEASED）。

标注 “<==3” 和 “<==4” 的行显示出container1;1.2访问的其他方式。

服务端触发器

基本解决方案的最后一个主要组件包括了一个与“tag”命令事件相关的服务端触发器。该触发器安装在托管IP块的SyncServer上。该触发器在标签名称与预定义命名约定匹配时将搜索“tag”事件（在范例中，标签必须与REL*的值匹配）。找到匹配的标签后，若该对象被定义为模块，则可以获取到其“whereused”模块层级。若该模块的任何父级与预定义容器模块的名称匹配（在范例中容器的名称必须与container*值匹配），则该触发器将添加“tag transaction”到镜像系统中。被添加的事务与特定的容器模块关联。当镜像系统发现“tag”事务时，它将把容器模块与容器镜像匹配起来，并用相关镜像通知每个MAS升级容器。

该触发器是用于减轻SyncServer可能处理针对各版本控制行为的成千上万镜像定义导致的不必要的负荷。该触发器仅搜索并处理“tag”命令事件。若标签与模块并不相关，或标签与发布块命名约定不匹配，则该触发器就会存在。

为配置服务端触发器，请先在SyncServer上启用“tag”版本控制的注释创建。可以通过Administrater Server Server Settings控制面板选择RC注释页。详见图10。

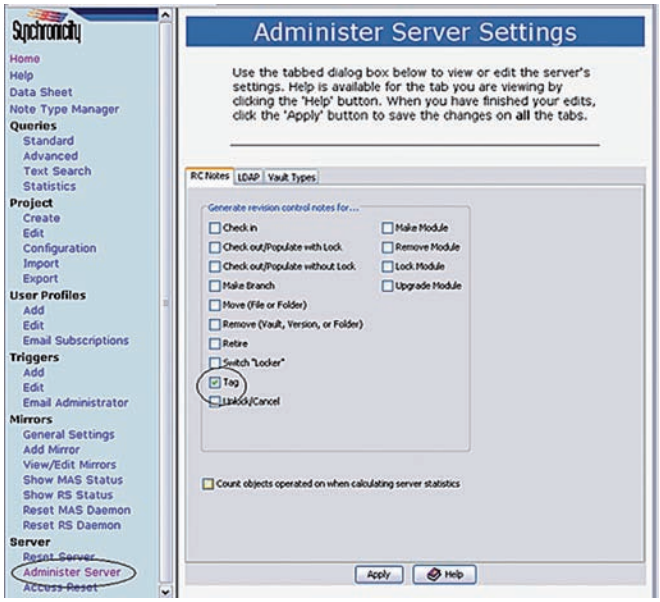

图 10

本文附有范例TCL文件，可以运行其代码安装服务端触发器。可以通过自定义代码配置预定义标签名称和容器模块名称。请将触发器代码文件放在SYNC_CUSTOM_DIR/servers/<host>/<port>/share/tcl目录，并通过TriggerAddNoteActive控制面板安装触发器。详见图11。

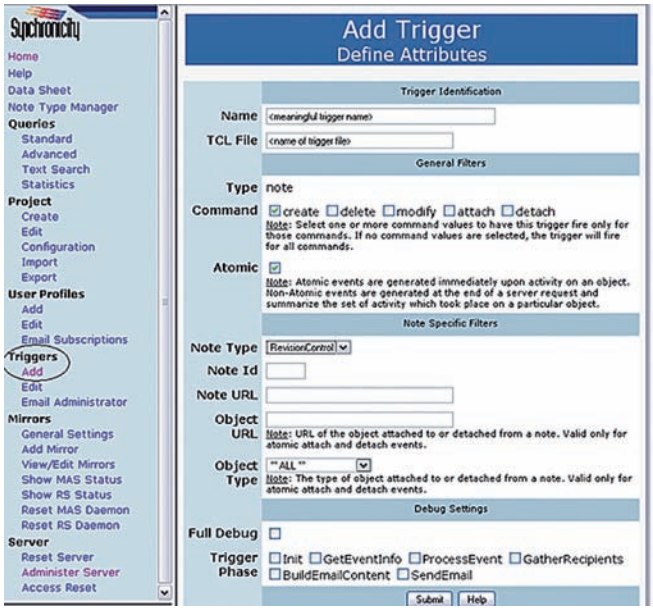

图 11

处理多级块的变更

之前介绍的基本解决方案将<Block>;RELEASED模块推送到每个设计中心，其容器模块包含到RELEASED模块的分级引用。若该块之下的模块层级发生变化，这些块的变更必须通过以下过程自动推送到设计中心。

图12中的container2模块展示了模块的层级。

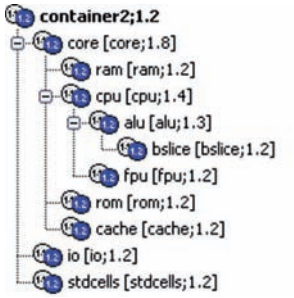

图 12

容器模块仅有到core;RELEASED，io;RELEASED和stdcells;RELEASED的分级引用。当这三个块中任何一个的RELEASED标签移动到模块的新版本中，则重新标记将导致触发器触发，最终导致镜像递归获取container2模块。

我们考虑一个范例，有一个新的版本“ram”被标记为“RELEASED”。从“core”开始的分级遍历将解除到ram;RELEASED的分级引用，并查看被标记的“ram”的新版本。然而，镜像从container2;Trunk:Latest运行了“populate-recursive”。当遍历访问到core;RELEASED时，它将切换到静态hrefmode。一旦实现切换，它将总是使用保存在指向“ram”分级引用中的“ram”的静态版本（在这里仍然是ram;1.2）。仅仅为“ram”的新版本标记“RELEASED”标签不会直接导致container2的“populate-recursive 获得ram;RELEASED。

获取“ram”的新版本需要有“core”的新版本进入并标记为RELEASED。通过“core”的进入，“ci”命令更新了在“ram”分级引用中指向从“core”到“ram”的静态版本（假设当“core”进入时，“core”和“ram”在同一个工作区域）。

如果你想要查看“core”的新版本是否需要在“ram”发布后进入，请在开发中存放块的工作区域运行“showstatus-recursive 命令。详见图13。

```
stcl> showstatus -rec -report brief core%0
core%0: Version is Up-to-date
core%0: Hierarchical reference conflicts found.
HREF: ram
    EXPECTED: sync://qewfsun9:30036/Modules/ram, selector=RELEASED, version=1.2
    ACTUAL:   sync://qewfsun9:30036/Modules/ram, selector=RELEASED, version=1.3
core%0: Hrefs are Out-of-date
...
core%0: Module hierarchy is Out-of-date.
core%0: Module is Out-of-date.
core%0: Needs update.
core%0: Needs checkin.
```

图 13

“showstatus”输出清晰地显示出到ram;RELEASED的“core”的分级引用期望找到RELEASED标签来更改到1.2版本，而不是1.3。这意味着“core”需要进入来同步静态版本，保持其对“ram”的分级引用，一边与工作区域中的“ram”版本匹配。

“core”新版本的简单进入无法实现向设计中心的必要推送。它必须将其RELEASED标签移动到刚进入的新版本上。详见图14。

```
stcl> tag -replace RELEASED [url vault core]\:[url versionid core]
Tagging:     sync://iphost:80/Modules/core:1.9 : Added tag 'RELEASED' to version '1.9'
```

图 14

更详细地说：若“alu”块的新版本已发布，则“cpu”可能需要进入并被标记为RELEASED，“core”同样也是。

在范例中，若“ram”、“alu”或“cpu”的RELEASED标签移动到其块的更新模块版本上，这将依次导致触发器在“container1”上添加虚拟标签事务。与“container1”关联的设计中心镜像将获得块的每个已发布版本的更新。

使用镜像块

正如上文所述，-mirror选项不适用于模块，因此将模块自动更新到镜像不能使用“populate-mirror”。但是一旦镜像通过使用“populate-share”获得更新，所有模块成员文件就存在文件缓存中。希望使用缓存中模块的个人用户就可以使用以下方式来实现：

- 使用“populate-share<到模块的URL>”的只读方式获取模块（或其整个层级）。取出到工作区域中的每个模块的成员文件将成为指向文件缓存中相应成员的符号链接。可以作为完整副本（populate-get）获取独立成员文件，成员可被锁住（populate-lock）以获得文件的可编辑版本。
- IP块的只读引用副本存放在设计中心的镜像中。他们不会在其工作区域内被直接复制或使用。相反，设计中心使用的工具与设计流程期望无需通过进一步处理找到镜像目录中的块，而不是通过镜像更新过程获取自动更新。
- 如果使用硬链接和（或）模块缓存（mcaches），可获得更好性能和更高效使用的系统资源。这些属于系统的扩展功能。

扩展

跟踪每个块的不同版本

在前述的简单模式中，已发布块只有一个版本被容器引用且推送到设计中心，但设计中心可能需要块的更多不同版本。这里介绍几种方法可将目前的机制扩展为跟踪每个块的不同版本：

- 采用命名约定的块使用标签名称区分质量等级。例如，GOLD、SILVER、BRONZE等标签名称常被使用。这将已经讨论过仅有的一个已发布标签扩展到了3个标签。当块的新发布版本可用时，可以运行“tag-replace”命令移动BRONZE标签到新发布版本。BRONZE版本是最新发布且最不稳定的版本。之前标记为BRONZE的版本则变更为SILVER，标记为SILVER的版本则变更为GOLD，标记为GOLD的标签则失去GOLD标签。它可能使用其他标签名称以注明是之前发布的版本。

为了扩展系统以为每个块处理多种可用的标签，必须更改服务端触发器来扩展其搜索的已知标签名称，以包含匹配GOLD*, SILVER*和BRONZE*的标签名称。

另外，<Block>的三个分级引用<Block>;GOLD,<Block>;SILVER and <Block>;BRONZE必须添加到容器模块中。在这三个分级引用中的相对路径需要在生成3个块版本的镜像工作区域中更新到特定的目录名称。相同块每个版本的相对路径可以被定义为：“../<BlockName>/<Selector>”或某些变量。

● 除了使用动态标签名称结构，如GOLD,SILVER和BRONZE，还可以为模块的每个已发布版本分配一个唯一的标签，该标签在块的新发布版本创建时不会移动。此类标签名称的范例如REL1.0, REL2.0和REL3.1。

因为每次在块的新版本发布时，必须照例将新的分级引用添加到每个容器中，这种做法会显得很麻烦。另一种更好的做法是可以通过重构服务端触发器脚本，配置自动将分级引用添加到容器中去。该脚本编写为仍然在服务器上处理每个标签事件，但其标签匹配所需的REL*模式，且在SyncServer上的每个容器模块都可以找到其分级引用（通过"showhrefs-format list"）。它被配置为搜索被触发的指向模块的分级引用（与分级引用所用的选择器无关，仅匹配基础模块名称）。若找到分级引用，脚本将通过将分级引用定义中的新标签作为选择器，从容器中添加分级引用到新发布的模块版本。由于从容器到新块的分级引用并不存在，在创建并发布新块时问题并不能得到解决。触发器可以在容器内为新发布的块创建新的分级引用，但是因为每个容器只保存设计中心感兴趣的块的分级引用，这其实并不让人满意。每个设计中心不一定对新的块都感兴趣，因此可能需要从容器模块中手动添加分级引用。

使用GOLD、SILVER、BRONZE扩展后，在新增分级引用中定义的相对路径必须为唯一，以避免在镜像工作区域的相同基本目录中重复带有不同选择器的相同块。

● 可以为特定的设计中心定义带不同逻辑、用以处理不同容器的服务端触发器。服务端触发器受管理员控制，用于为SyncServer托管IP块。并且可能存在多个SyncServer托管来自多个设计团队的块。系统支持复杂运算以便为块处理多种发布模型和发布机制。

设计中心的高效数据利用

首选解决方案通过调用"populate-share"将块放到镜像中去，因此所有数据文件都存放在文件缓存中，并可从用户工作区域链接过来。我们还可以进一步优化文件系统资源以通过使用块大幅提高性能。

使用硬链接取代符号链接

用户可以在Unix工作区域的文件缓存中创建硬链接取代动态符号链接。符号链接的缺点是每个链接都需要一个索引节点（一种文件系统资源），而每个指向同一文件的硬链接不会占用另一个索引节点。

例如，若DesignSync文件缓存中的一个文件有指向它的3个符号链接（其中一个来自于镜像的模块，另两个来自于用户工作区域），则文件系统共需要为该文件使用4个索引节点（其中一个用于缓存中的文件本身，另三个分别用于每个符号链接）。若DesignSync使用硬链接，在该例中仅需要使用一个索引节点。缓存、镜像及两个用户工作区域中的每个文件都拥有指向磁盘或操作系统管理的相同文件内容的直接引用。所有这些直接引用仍然只用一个索引节点来链接文件。

使用硬链接的关键要求是所有使用硬链接的地方必须存放在同一个文件分区且在相同的挂载点下。因此，在我们的范例中，当文件缓存、镜像工作区域及用户工作区域均位于同一个文件分区且使用相同的挂载点（可能存放于某个大型文件服务器上）时，我们可能获得最高的运行效率。

若要使用硬链接，请使用SyncAdmin访问Site OptionsLinks控制面板，勾选"为工作区域创建到缓存文件的硬链接"复选框。见图15。

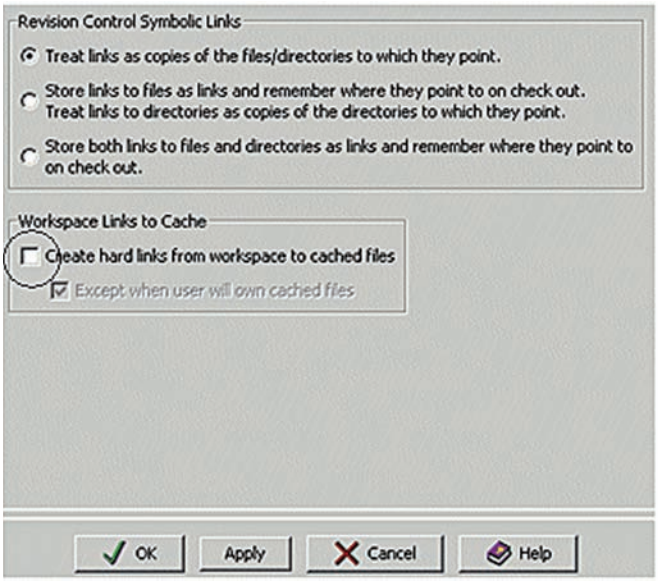

图 15

DesignSync的智能处理允许通过"populate-share"检查所使用的工作区域是否与文件缓存在同一个分区与挂载点上。若符合要求且"为工作区域创建到缓存文件的硬链接"复选框已被勾选，系统就可以自动创建硬链接。例如，若用户锁定文件用于修改，DesignSync也会适时断开硬链接以便完全写入文件副本。若工作区域和文件缓存并不在同一个分区和挂载点上，即便SyncAdmin中的复选框已被勾选，DesignSync仍将返回动作，转为创建符号链接。请注意创建硬链接的复选框默认是为不勾选的。在DesignSync中可能需要进行更多的检查，因此是否默认创建硬链接的选项将由管理员来决定。

使用SyncAdmin将在SiteRegistery.reg注册表键上定义硬链接。因此创建硬链接将应用到所有已安装DesignSync的客户端。控制硬链接创建的注册表键可以在PortRegistery.reg或MirrorRegistry.reg中手动更改。需要设置的键为：

- HKEY_LOCAL_MACHINE\Software\Synchronicity\Client\Cache\PBFCEnabled=dword:1
- HKEY_LOCAL_MACHINE\Software\Synchronicity\Client\Cache\PBFCAllowUserToOwnFile=dword:1

配置并使用模块缓存（mcache）

要想提高在每个设计中心自动更新块的效率，另一种方法是将镜像配置为模块缓存（mcache）。

使用mcache最大的好处是如果"populate"发现了正在搜索的模块特定版本已存在mcache中，则系统将在工作区域中创建单一的符号链接并在mcache中指向到模块的基本目录。在mcache中的模块可能有大量成员文件，由于都位于工作区域中，我们可以很容易访问这些文件。这是因为我们可以将工作区域的符号链接视同为mcache模块的基本目录位于工作区域内一样。在用户工作区域内仅创建一个到模块基本目录的符号链接比创建模块每个成员文件的符号链接（或硬链接）或获取其完整副本要快的多。

将镜像配置成mcache非常简单。若该镜像的顶层目录已运行了"setroot"，则它已被声明为一个工作区域的根目录。这足够使之可以存放获取到镜像内所有模块有关的元数据。因此一般来说任何工作区域都可以作为mcache。

用户可以使用"populate-mcachepath"来搜索在某个mcache内的模块或在不同mcache中的大量模块。该选项可指定一个或多个指向工作区域的路径名称作为mcache。"populate"会在基于第一次在服务器上决定模块选择器的mcache中搜索其所需的特定模块版本。

使用mcache也需要考虑一些情况：

- 若"populate"以递归方式获取模块的层级，则必须以递归方式获取mcache模块才能使层级中的模块与"populate"搜索匹配。
- 任何在mcache中拥有子模块级别的模块都应使用基于"../<path>"而不是"../"的相对路径获得。在工作区域中创建mcache符号链接时，由于命令无法访问子模块，"populate"命令就无法识别是否使用了基于"../"的相对路径来获取这些子模块。一旦在mcache中创建了模块的符号链接，"populate"就在模块层级中停止搜索。创建包含在模块层级中、基于模块目录的对等目录结构（通过使用".../"相对路径指示在层级中的模块）很可能将导致使用mcache出现问题。用户工作区域仅可以获得指向mcache中顶层模块的符号链接。因为子模块不会存留在工作区域中符号链接指向的基本目录下，用户区域中很可能将无法访问子模块。

- mcache中的模块数据通常是只读的，无论是因为mcache/镜像以SUID模式创建，还是被不同用户获取。因此不仅模块成员的文件是只读的，甚至通常在mcache中模块成员文件间无法创建额外的数据文件。所以mcache只能成为块的只读引用快照。当然，若使用"-share"模式填充mcache/镜像，则单独的模块成员文件就存在文件缓存中，且可以使用"-share"选项填充用户工作区域来获得指向成员文件的符号链接（或硬链接）。这种配置允许用户工作区域拥有创建在指向模块成员文件链接的额外数据文件。最终，用户工作区域可以填充"-get"或"-lock"选项来获得模块成员文件的完整副本。在这种情况下，成员文件的检入（若访问控制允许）将在托管块的仓库SyncServer上创建新的模块版本。重新标记块的新模块版本会是一个问题，即是否发布刚进入的变更。

为了获得最高的效率，用户可以配置镜像来使用"populate-share-mcachepath<mcacheDirectory>"并配置MAS使用硬链接。"-mcachepath"选项允许在缓存中自行创建到缓存模块的符号链接。所有成员文件的单一版本都存在文件缓存中，且每个模块的单独成员文件通过硬链接指向到文件缓存中的成员文件。这种做法可以最优化磁盘空间和索引节点的使用，使得仅需要为文件缓存中的每个特定成员文件版本使用一个索引节点。由于使用了符号链接指向缓存模块的基本目录（见侧边栏），缓存中的目录节点就可以控制在最少数量。当缓存中拥有较多的不同模块层级且在层级中使用的许多不同子模块都是单一模块的相同版本时，这种配置就显得非常有用。

文件缓存和模块缓存间的优化是如何产生的

在某个设计中心，一个设计师可能使用块的不同版本A、B和C，而另一个设计师可能使用版本B、D和F。如果他们都使用相同的版本B，由于第一个用户已经调用了版本B，使得它被复制到文件缓存中，后面的用户就可以直接从本地获取文件而受益。

优势还体现在对文件缓存和模块缓存的操作中。文件缓存包括各自文件的副本。模块缓存将使用一个容器对象（模块）来代表对象的特定版本。模块是一个抽象的对象，如块。例如，可能有一个模块代表由成千上万个文件组成的CPU对象。模块缓存可以很简单地链接到文件缓存中组成某一版本对象的文件的特定版本，既可以降低磁盘空间的占用，又可以大幅减少服务器与设计中心文件传输的时间。

访问正在开发的块

如果需要访问一个正在开发的块，最简单的方法是创建一个正在开发的块的单独镜像并使用"<branch>:Latest"选择器来对模块的变化进行分支跟踪。每次这个块一旦有版本控制操作发生，镜像就会推送该更新。该简单的解决方案可以使正在开发的块保持在每个设计中心的更新，但是必须跟踪大量的镜像定义——这样的结果并不是我们所期望的，上文已经详述了这种结果产生的原因。

处理这种情况更好的方法是用另一个服务端触发器创建另一个容器模块的组合。容器模块将仅包含到正在开发的块的分级引用（例如到每个块的"<branch>:Latest"选择器）。新的服务端触发器将在"checkin"事件在模块上发生时进行替代（代替"tag"事件）。该触发器取代进入的模块并仍然遵循whereused反向引用来寻找相关容器的模块。随后检查容器模块的分级引用（通过"showhrefs-format list"）查看"<branch>:Latest"选择器已经进入的模块的分级引用。若找到匹配的分级引用，则"tag"事务将被添加到服务器镜像事务日志中，这些日志显示镜像填充容器并获取新增到正在开发的块的内容。

使用自动生成的镜像

借助DesignSync的"自动生成镜像"扩展（也称之为"Scripted Mirrors"），镜像系统可以在满足一定的条件下自动更新镜像。该条件以脚本的形式进行约定并完全支持自定义。处理大部分使用情况的脚本是随着块的发布而提供的。一旦脚本启动一个自动生成的镜像，该镜像将获取其所需的数据。该镜像将保持相对较短时间（如一天）的活动状态。若在跳转URL及与生成的镜像关联的选择器上没有任何活动发生，则该镜像将被自动禁用。当条件符合时，该镜像又将重新生成。由于在指定的时间内只有少量镜像被激活，该系统允许镜像系统处理大量镜像。

以下是如何使用自动创建的镜像更简单地解决上述IP块分布式系统问题的方法：

- Add Mirror控制面板的扩展允许当创建自动生成的镜像使跳转选择器指定通配符（见图16中#1标记处）。因此REL*的选择器可能被指定用于URL "sync://iphost:80/Modules"。这个配置过程导致在iphost:80上的每个包含起始于"REL"的模块都可能被镜像。当创建自动生成的镜像（#2标记处）时，所提供的TCL脚本在MAS上运行并决定是否获取每个模块。

如果使用了标签命名规则如GOLD, SILVER和BRONZE，由于没有通配符匹配所有三个选择器，我们必须使用其他方法来解决问题。这些选择器可能被替代为"*"（所有匹配的选择器）且脚本看起来仅针对标记了GOLD,SILVER和BRONZE标签的镜像的操作，使得可以获取这些模块。另一个方法是可以创建三个不同的脚本镜像：一个使用GOLD选择器，一个使用SILVER选择器，一个使用BRONZE选择器。

● 容器模块并不是用来定义将IP块模块推送到设计中心的必须条件。相反，每个设计中心创建起自己的文件列出需要从仓库服务器获取的IP块清单。TCL脚本则读取这份清单文件并确定该模块及其选择器是否应被镜像。该清单文件可以是期望的任何文件格式。一个可能以程序设定方式运行的"cron"进程将确定IP仓库中哪个IP块将要被推送到设计中心并更新列表文件。

另一种方法可能仍是利用顶层模块定义具有IP块的模块层级。通过在该顶层模块运行"showhrefs-rec"，该列表文件随后将被得到定义。在每次需要执行命令确定模块是否应该被取出时让TCL运行"sharehref-rec"脚本是不合适的。

● 服务端触发器将不再需要。在以前，该触发器寻找标签事件并搜索模块的"whereused"层级来确定该容器是否应被填充。现在已生成镜像的TCL脚本可以读取列表文件来确定应被取出和应被忽略的模块。

● 由于不再有容器模块存在，用户无需更改任何注册表配置来避免容器创建子镜像。

● 在镜像运行时，用户无需手动定义注册表配置使镜像系统运行"populate-share"。现在可以使用Add Mirror控制面板来定义镜像是否可以使用"-share"（图16中#3标记处）。

● 当镜像运行"populate-share"时，Add Mirror控制面板也允许对镜像的简单配置来使用硬链接取代动态符号链接（图16中#4标记处）。

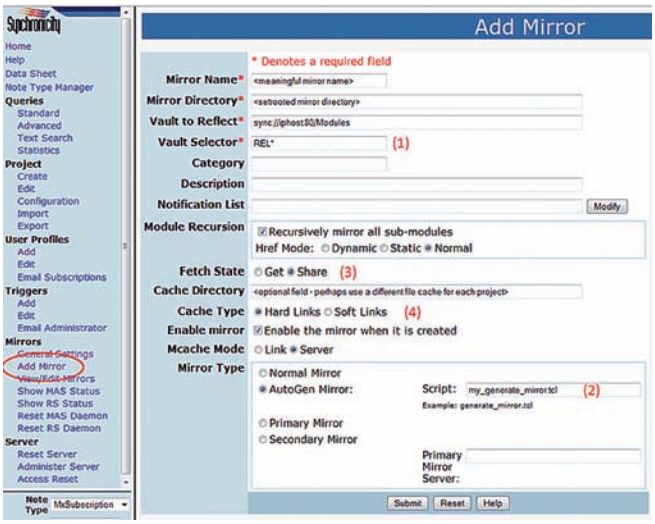

图 16

总结

DesignSync提供了一系列组合机制可以用来创建全球设计团队所需的功能强大的数据管理解决方案。由模块和分级引用定义的高级抽象概念完美适用于许多大型集成电路（IC）及SoC设计的基于块的设计范例。镜像系统支持自定义，使得大型数据可以在不同地点的不同设计中心间快速简单地得到复制。在用户需要用到数据之前将数据更新到本地，可以实现对已发布设计数据快速优化地访问。最终，借助DesignSync的文件缓存、符号链接、硬链接及模块缓存可以优化用于所有数据的系统资源。在复杂的IC设计过程中，部署可以扩展到跨多项目的高性能、协作性设计环境的能力成为成败的关键要素。DesignSync是用于解决多站点IP重复使用设计需求的先进且验证可靠的解决方案。

附录 – TCL代码

```
#
# set up the trigger as
#
# NAME:  <desired name>
# Tcl File: <name of  the file>
# place the file in custom/servers/<host>/<port>/share/tcl
#
# Active : yes
# Command:  'create'
# Atomic:  yes
# Note Type: RevisionControl
# ObjectType:  ** ALL **
#

namespace eval ::NotifyMAS {

        #-------------------------------------------------------------
        # checkSanity: check that the required params are present
        #-------------------------------------------------------------
        proc checkSanity {} {
                if { ! [info exists ::SYNC_Parm(tag)]  } {
                        return 0
                }

                if { ! [releaseTag $::SYNC_Parm(tag)] } {
                        return 0
                }

                if { ! [info exists ::SYNC_Parm(objects)]  } {
                        return 0
                }

                return 1
        }

        #-------------------------------------------------------------
        # getModUrl: Get the first object from the 'objects' list
        #                 On a tag op, this is a version url, get the
        #                 module url, by dropping the ;<rev>
        #-------------------------------------------------------------

        proc getModUrl {} {
                set mv [lindex $::SYNC_Parm(objects) 0]

                # remove the host/port and version part
                set mp [url path $mv]
                set idx [string last ";" $mp]
                if { $idx > 0 } {
                        incr idx -1
                        set mp [string range $mp 0 $idx]
                }
```

```
        set murl sync://$mp
        return $murl
}

#-------------------------------------------------------------
# isContainer:  Check if given module url is a container
#-------------------------------------------------------------
proc isContainer {mod} {

        ############################################################
        ############################################################
        # CUSTOMIZE: this is where to customize the set of
        # modules that the set of submodules to distribute to
        # each remote site.
        ############################################################
        ############################################################
        set clist [list container*]
        set leafname [url leaf $mod]
        foreach m $clist {
                if { [string match $m $leafname] } {
                        return 1
                }
        }
        return 0
 }

#-----------------------------------------------------------------
# notifyCheckRecursive:  Check the whereused by looking
#                        up the back ref, do it
#                        ecursively When a match against
#                        container is found, add to array
#-----------------------------------------------------------------


proc notifyCheckRecursive {mod} {
        # get the back refs, and check for macro top level
        # if found, then break out...else reach end

        set backrefs ""
        if { [catch {url getprop $mod SyncBackRefs} backrefs] } {
                return
        }

        foreach br $backrefs {
                if { [isContainer $br] } {
                        # the mirror tag transaction needs a URL with a version
                        # number, so just use a plain 1.1 version. it's not
                        # really impacting anything.
                        set br "$br;1.1"
                        set ::foundArray($br) 1
                } else {
                        # recurse on the back ref
                        notifyCheckRecursive $br
                }
        }
}
```

```
#-------------------------------------------------------------
# processFound:  Process the array by
#       adding transactions
#       Add a tag transaction on the container
#       with the appropriate tag name
#-------------------------------------------------------------

proc processFound {tagval} {
        foreach u [array names ::foundArray] {
                if { $::foundArray($u) } {
                        # puts "_translog append [list $u "tag" $tagval 0 [list from trigger]]"

                        # this is the internal API to add a transaction to the Repository
                        # Server's MPD, and that gets pushed to the appropriate mirrors
                        # registered against the URL. in this situation, we want to say
                        # that the container module has been tagged so that it gets auto-populated
                        # into the mirror (the container really wasn't tagged).
                        _translog append [list $u "tag" $tagval "" [list from trigger]]
                }
        }
}

#-------------------------------------------------------
# releaseTag:  Check if its a release tag
#-------------------------------------------------------

proc releaseTag { tagval } {
        ##########################################################
        ##########################################################
        # CUSTOMIZE: This is where to define the set of tags
        # to look for to determine whether a submodule should
        # cause it's "whereused" chain of backrefs to be
        # searched for a container module.
        ##########################################################
        ##########################################################
        set relTagsList [list REL*]
        foreach t $relTagsList {
                if { [string match $t $tagval] } {
                        return 1
                }
        }
        return 0
}

#-------------------------------------------------------
# isModule:  check if module
#-------------------------------------------------------
proc isModule {item} {
        set idx [string first /Modules/ $item]
        if { $idx < 0 } {
                return 0
        }
        return 1
}
```

```
        #-------------------------------------------------------
        # main:  start and complete all the processing
        #-------------------------------------------------------

        proc main {} {
                if { [checkSanity] } {
                        catch {
                                # the tag value
                                set tagval $::SYNC_Parm(tag)

                                # is it a release tag
                                if { ! [releaseTag $tagval] } {
                                        return
                                }

                                # array to hold results
                                array unset ::foundArray
                                set ::foundArray(dummy) 0

                                # get the module url
                                set modurl [getModUrl]

                                # is it a module
                                if { ! [isModule $modurl] } {
                                        return
                                }

                                # check back refs recursively
                                notifyCheckRecursive $modurl

                                # process any found
                                processFound 1,SyncBranch#$tagval

                                # puts [array get ::foundArray]
                        } msg
                        # puts $::errorInfo
                        puts $msg
                }
        }
}

::NotifyMAS::main
```

DASSAULT SYSTEMES

提供最佳产品

CATIA
虚拟产品设计

ENOVIA
全球协作生命周期管理

SOLIDWORKS
面向专业的3D技术

EXALEAD
信息智能

SIMULIA
真实仿真

3DSWYM
社会创新

DELMIA
虚拟制造

3DVIA
3D在线逼真体验

作为一家3D体验公司，达索系统为企业和客户提供虚拟空间以模拟可持续创新。其全球领先的解决方案改变了产品在设计、生产和技术支持上的方式。达索系统的协作解决方案更是推动了社会创新，扩大了通过虚拟世界来改善真实世界的可能性。集团为80多个国家超过15万个不同行业、不同规模的客户带来价值。如欲了解更多信息，敬请访问：http://www.3ds.com/。

欧洲/中东/非洲
Dassault Systèmes
10, rue Marcel Dassault
CS 40501
78946 Vélizy-Villacoublay Cedex
France

亚太
Dassault Systèmes
Pier City Shibaura Bldg 10F
3-18-1 Kaigan, Minato-Ku
Tokyo 108-002
Japan

美洲
Dassault Systèmes
175 Wyman Street
Waltham, Massachusetts
02451-1223
USA

品牌网址
3DS.COM/ENOVIA

WI-ESDDM1205