JS 引擎调研与优化报告

张昱 (yuzhang@ustc.edu.cn) 课题组

中国科学技术大学计算机科学与技术学院

January 11, 2019

目录

Part I

组会篇

1 2018

1.1 秋季学期

参与人员: **张昱、霍姚远、王瑞凯、张恺奕、黄奕桐、张劲暾、冉礼豪、刘时、唐凯成**
2018 年 10 月 19 日。
分组:
JS 引擎结构: **张恺奕、黄奕桐**
龙芯架构: **刘时、冉礼豪**
GC 算法: **张劲暾、王瑞凯、唐凯成**
参考书:

1.1.1 2018.10.25 组会

出席人员: **张昱、余银、霍姚远、王瑞凯、张恺奕、黄奕桐、张劲暾、冉礼豪、刘时、唐凯成**
记录: **张昱、王瑞凯**

龙芯余银来介绍

- 引擎发展
  1. SpiderMonkey
  2. v8
  3. wsam
- SpiderMonkey 测试
  1. jit-test — 正确性测试
  2. speedometer、octane2 — 进阶正确性测试
  3. user application — 个性化测试
- 龙芯的工作经验
  1. 不对齐的 double — 不对齐指令的本身缺陷与对齐指令的内核异常
  2. longjump — mips64: 48 位常量, 6 条指令
  3. 内存分配 — jemalloc, Q. 内存分配问题: gc 的内存分配是否还存在调优空间
  4. 乘法指令 — mul 代替 mult, mflo
- debug 建议
  1. debug 版本, -D 选项 — 触发次数: hot BBs
  2. Linux perf — 时间统计, 网站: perf-io

工作情况

- **张昱**: Q1. 问题:
- **霍姚远**:
  1. 观察 js 引擎的并发性
  2. 整理引擎启动的大致流程与可调参数
- **张恺奕、黄奕桐**
  1. js 引擎的基本结构, 包括 v8 和 firefox
  2. js 中的 shape、inline cache 和 prototype 的简要介绍

3. SpiderMonkey的 gc 实现简介

- **刘时、冉礼豪**
  1. 龙芯芯片 (3A300) 的结构与特点
  2. 龙芯芯片中的分级缓存结构
- **张劲暾、唐凯成、王瑞凯**
  1. 常见的 gc 算法原理与问题
  2. SpiderMonkey中的 gc 实现与问题

下步工作建议

- **张昱**:
- **霍姚远**:
- **张恺奕、黄奕桐**
  1. 结合源码深入调研 shape、inline cache、prototype chains 的相关原理
  2. 了解 v8 引擎结构
  3. 关注 inline cache 与线程相关的部分
- **刘时、冉礼豪**
  1. 在手册阅读工作的基础上对龙芯架构与线程调度相关的内容，从硬件到操作系统，进一步调研
  2. 分析龙芯多核处理器核与核之间、核与共享缓存之间的网络结构
  3. 学习 Linux perf 的使用，探究硬件特性在操作系统中的体现
- **张劲暾、唐凯成、王瑞凯**
  1. 汇总分析SpiderMonkey源码中 gc 部分的数据结构与算法
  2. 关注 gc 部分的多线程实现细节和冲突处理方法
  3. 关注 gc 中块的分配 (malloc) 与回收 (finalize) 中的潜在问题与优化方向

1.1.2 2018.11.07 安装机器

出席人员：**张昱、余银、霍姚远、王瑞凯、张恺奕、黄奕桐、张劲暾、冉礼豪、刘时、唐凯成**
记录: **张昱、王瑞凯**

编译

release 版本

执行以下命令：

```
- cd js/src
- mkdir build_release
- autoconf
- cd build_release
- ../configure --enable-release
```

debug 版本

待补全
另外，端口号 3421 的机器上已经有了 release 和 debug 两个版本的编译结果

在 debug 版本下的测试指令

- `cd build_debug`
- `jit_test/jit_test.py ../build_debug/dist/bin/js` (上传社区时增加选项 –tbpl)
- **octane** — `../build_debug/dist/bin/js run.js`
- js 选项 — -D dump 字节码
- js 交互式命令
  - dis(sum) — 反汇编
  - IONFLAGS=options 传给 ion 编译
  - `export IONFLAGS=help` — 尽在 debug 版本生效

Fedora 软件安装

Q. 我记得是 Fedora，待验证

`yum install / yum search`

相关文件 — /etc/yum/

1.1.3 2018.12.17 工作安排

出席人员：**张昱、霍姚远、王瑞凯、张恺奕、黄奕桐、张劲暾、冉礼豪、刘时、唐凯成**

记录：**刘时**

待办事项

1. 在端口号 3422 的机器上安装 Mozilla 和 V8，并记录下载、编译过程中遇到的问题
2. 保证两台机器的远程桌面可用并验证通过
3. 基于`firefox-52.3.0`建立新 Repo 分支，起名`firefox-52.3.0-work`
   - 制作 release 和 debug 两个版本
   - 阅读完整的即时编译测试脚本`jit_test.py`，明确原理，在其基础上新建文件，构建新的测试脚本
   - 阅读 benchmark 脚本`${mozilla_root_dir}/js/src/octane/run.js`，对 octane 进行调研
   - 调研 js 运行选项：某些选项与 GC 相关，分析在使用不同选项时输出的区别
   - 新建 git repo，代替`firefox-52.3.0-work`分支
4. 对 octane 中的一个程序（可新建文件，基于 run.js 作修改），尝试使用 perf 分析其性能表现，学习 perf 的使用。

(上述各项工作过程中可能用到的命令和参考资料见`${repo_root_dir}docs/meetingSupplement/memo12172018.md`)

Q&A

Q：目前文档比较凌乱，应该如何整理?

A：接下来可以尝试使用 Overleaf 进行文档的在线协作。

Q：运行 benchmark（jit_test，octane，etc）的输出看不懂怎么办?

A：问问题：先整理出自己做了什么（运行的命令），提供输出内容，指出哪里看不懂，再与他人交流；自己研究：阅读源代码，看注释或用 print，分析程序的功能。

Q：龙芯手册内容不全怎么办?

A：直接在微信群里与龙芯的人交流。

Q：baseline 中具体调用 ic 的地方还有 baseline 入口点

A：baseline 有一个 pass 的名字，这里关注的是 js 虚拟机核心的代码，里面启用了 jit 装代码，里面判断是走 baseline 还是其他。基本的虚拟机代码结构：按方法编译，针对每一个方法先找 native code，如果没有再启用编译自己的代码，这段代码一般在 vmcore 上。待完善

Q：如何通报自己目前的工作?

A：每位同学在确定了自己下一步的工作计划后，就在群里告诉大家，这样一方面可以让工作相关的同学们更好的写作，另一方面也可以避免大家做重复的工作。

Part II

日志与环境篇

2 日志记录

2.1 20181217

启动 Overleaf 项目。**王瑞凯**、**唐凯成**和 **张劲暾**使用 wrk15835@mail.ustc.edu.cn，**冉礼豪**和**刘时**合用 fere0001@gmail.com，**张恺奕**和**黄奕桐**合用 hyt@mail.ustc.edu.cn

2.1.1 待办事项

1. 在端口号 3422 的机器上安装 Mozilla 和 V8，并记录下载、编译过程中遇到的问题
2. 保证两台机器的远程桌面可用并验证通过
3. 基于`firefox-52.3.0`建立新 Repo 分支，起名`firefox-52.3.0-work`
   - 制作 release 和 debug 两个版本
   - 阅读完整的即时编译测试脚本`jit_test.py`，明确原理，在其基础上新建文件，构建新的测试脚本
   - 阅读 benchmark 脚本`${mozilla_root_dir}/js/src/octane/run.js`，对 octane 进行调研
   - 调研 js 运行选项：某些选项与 GC 相关，分析在使用不同选项时输出的区别
   - 新建 git repo，代替`firefox-52.3.0-work`分支
4. 对 octane 中的一个程序（可新建文件，基于 run.js 作修改），尝试使用 perf 分析其性能表现，学习 perf 的使用。

(上述各项工作过程中可能用到的命令和参考资料见`${repo_root_dir}/docs/meetingSupplement/memo12172018.md`)

2.2 20190102

2.2.1 3421 端口的机器网络问题

问题：机器连不上网
原因：目前认为是网卡接触不良
解决方法：换个网卡

2.2.2 git 版本创建后的 push 问题

问题：[机器 3421] 本地创建了 firefox-52.3.0-work 分支，尝试执行以下命令：

```
git push --set-upstream origin firefox-52.3.0-work
```

得到的结果为：

```
fatal: http://cgit.loongnix.org/cgit/mozilla-esr52/info/refs not valid: is this a git repository
       ?
```

[**张昱**] 在课题组 git 服务器建了 mozilla-esr52 仓库并设置大家的访问权限. 可以将 loongson 仓库的原有分支和自定义分支存档到课题组自己的服务器的仓库中。

在另一个远程仓库（ustc）创建并复制当前仓库的分支 branchname 可以这样做:

```
git remote add ustc ssh://git@222.195.92.204:1422/mozilla-esr52
git push ustc branchname
```

[**王瑞凯**]Q.：龙芯的服务器上还是没有 push 的权限

A.：现在服务器上可以 push 了，已经将分支 firefox-52.3.0-work 推送到库里

2.2.3 3421VNC 服务器问题

[**王瑞凯**]Q.：3421 机器上的 ssh 没有问题了，但是 vnc 还是连不上

```
[loongson@localhost mozilla-esr52]$ sudo systemctl status -l vncserver@:1.service

  vncserver@:1.service - Remote desktop service (VNC)
   Loaded: loaded (/usr/lib/systemd/system/vncserver@.service; enabled)
   Active: failed (Result: exit-code) since 三 2019-01-02 18:16:39 CST; 6s ago
  Process: 6141 ExecStart=/sbin/runuser -l <USER> -c /usr/bin/vncserver %i (code=exited, status
        =1/FAILURE)
  Process: 6138 ExecStartPre=/bin/sh -c /usr/bin/vncserver -kill %i > /dev/null 2>&1 || : (code=
        exited, status=0/SUCCESS)

1月 02 18:16:39 localhost.localdomain runuser[6141]: runuser: 用户 <USER> 不存在
1月 02 18:16:39 localhost.localdomain systemd[1]: vncserver@:1.service: control process exited,
        code=exited status=1
1月 02 18:16:39 localhost.localdomain systemd[1]: Failed to start Remote desktop service (VNC).
1月 02 18:16:39 localhost.localdomain systemd[1]: Unit vncserver@:1.service entered failed state
        .
1月 02 18:16:39 localhost.localdomain systemd[1]: vncserver@:1.service failed.
```

[**刘时**]A.：把远程桌面重启一下

```
vncserver -kill :1
vncserver :1
```

3 VNC 安装与配置

3.1 20181217

[**刘时**]

vncserver 的安装与配置：

参考资料：https://www.server-world.info/en/note?os=Fedora_28&p=desktop&f=6

```
# 在 root 用户下
# 安装 tigervnc-server
dnf -y install tigervnc-server
# 设置防火墙，使之允许 vnc-server 的运行
firewall-cmd --add-service=vnc-server --permanent
# 重新加载防火墙使设置生效
firewall-cmd --reload
# 设置 vnc 连接的密码
vncpasswd
# 启动 1 号桌面，800*600 分辨率，24 位色彩
```

```
vncserver :1 -geometry 800x600 -depth 24
# 停止 1 号桌面的运行
vncserver -kill :1
# 修改桌面启动运行指令
# 将 “exec /etc/X11/xinit/xinitrc” 一行注释掉（用 ‘#’），改为 “exec /usr/bin/mate-session”
vi ~/.vnc/xstartup
# 重新启动 1 号桌面
vncserver :1 -geometry 800x600 -depth 24
# 查看当前正在运行的桌面
vncserver -list
```

远程桌面进度：在 3422 机器上试验，server 配置好了，可以正常运行，但是没法用 client 正常连接，macOS 下 VNC Viewer 和 TigerVNC Viewer 报错信息为：“Reading version failed: not an RFB server?”；Windows 下 UltraVNC Viewer 报错信息：“Connection failed：Invalid Protocol”

3.2 20181218

[**张昱**增加] - 增加了两台龙芯机器的 VNC 端口映射,

将 loongson1 的 5901-5909 端口映射到 222.195.92.204 的 15901-15909; 将 loongson2 的 5901-5909 端口映射到 222.195.92.204 的 25901-25909;

http://blog.itpub.net/7607759/viewspace-2133501/

- sudo yum install vnc-server

- sudo cp /lib/systemd/system/vncserver@.service /etc/systemd/system/vncserver@:1.service 这是设置 5901 端口的配置

- sudo vi /etc/systemd/system/vncserver@:1.service 主要做了两处改动，首先是将%i 替换为:1，其次是将 <USER> 替换为 loongson。这是指定 vncserver 运行在 loongson 用户下。后面运行出错 “Failed to start Remote desktop”，后将 loongson 改为 root，可以成功启动

- sudo systemctl daemon-reload 重新加载 systemd 的配置

- sudo systemctl enable vncserver@:1.service 启用这个服务，将 vncserver 服务设置为开机自启动

- 在 vnc 服务器上切换到 loongson 用户，执行 vncpasswd，设置访问密码

- sudo systemctl status vncserver@:1.service 查看服务状态

- Q: Failed to start Remote desktop(VNC)

[**刘时**]

远程桌面进度：现在两台机器的 loongson 账户中分别有桌面编号为 2 和 1 的两个远程桌面正在运行，大家可以使用 VNC Viewer 等 VNC 客户端连接访问（host：222.195.92.204；port：15902 或 25901）。大家也可以登录服务器，使用命令`vncserver -list`，查看正在运行的桌面，并进行增删（指令见这里），端口号的映射见这里。

3.3 20181218

[**刘时**]

远程桌面进度：目前两台机器上分别有 3 个虚拟桌面在运行，使用客户端连接的主机名和端口号分别为：(前三个在 222.195.92.204::3421 上，后三个在 222.195.92.204::3422 上)

- 222.195.92.204::15901
- 222.195.92.204::15902

- 222.195.92.204::15903
- 222.195.92.204::25901
- 222.195.92.204::25902
- 222.195.92.204::25903

大家可以使用VNC Viewer等客户端连接使用，密码为 loongson。如果需要修改桌面配置，请连接服务器并使用`vncserver`命令进行操作，指令见这里。

目前远程桌面的配置方法下需要修改 `/.vnc/xstartup`，修改的内容参考这里，但在一个账户下只需一次修改即可，如无必要请勿修改配置文件。

Q: 目前两台机器的 15904-15909, 25904-25909 端口建立的远程桌面无法使用客户端连接，因此最多只有 6 个远程桌面可用

Part III

Mozilla Firefox 篇

4 mozilla esr52

4.1 mozilla esr52 的下载

[冉礼豪]

- 时间:2018.12.17
- 机器:3422
- 地址:222.195.92.204::25901
- 执行命令:git clone git://cgit.loongnix.org/browser/mozilla-esr52.git
- 路径: /home/loongson/loongson/mozilla-esr52

4.2 mozilla esr52 的编译

[冉礼豪]

- 时间: 待定
- 问题:
  在/home/loongson/loongson/mozilla-esr52/js/src 目录下, 执行 autoconf 命令
  报错:../../build/autoconf/acwinpaths.m4:10: error: defn: undefined macro: AC_OUTPUT_FILES
  ../../build/autoconf/acwinpaths.m4:10: the top level autom4te: /usr/bin/m4 failed with exit status: 1

[张劲暾]

- 时间：2019.1.10
- 机器：3422
- 地址：222.195.92.204::25901
- 问题：将 mozilla 改为 ustc 库中的版本

```
git pull ustc firefox-52.3.0-work
//在/home/loongson/loongson/mozilla-esr52/js/src下
$ autoconf
../../build/autoconf/acwinpaths.m4:10: error: defn: undefined macro: AC_OUTPUT_FILES
../../build/autoconf/acwinpaths.m4:10: the top level
autom4te: /usr/bin/m4 failed with exit status: 1
$
//从3421机上copy文件js/src/configure后
[loongson@localhost src]$ cd build_debug/
Creating Python environment
New python executable in /home/loongson/loongson/mozilla-esr52/js/src/build_debug/
    _virtualenv/bin/python2.7
Also creating executable in /home/loongson/loongson/mozilla-esr52/js/src/build_debug/
    _virtualenv/bin/python
Installing setuptools, pip, wheel...done.
Error processing command. Ignoring because optional. (optional:setup.py:python/psutil:
    build_ext:--inplace)
Reexecuting in the virtualenv
checking for a shell... /usr/bin/sh
checking for unistd.h... yes
checking for nl_types.h... yes
...
checking for rpmbuild... /usr/bin/rpmbuild
```

```
    checking for autoconf...
    ERROR: Could not find autoconf 2.13
    [loongson@localhost build_debug]$ make
    make: *** 没有指明目标并且找不到 makefile。 停止。
    [loongson@localhost src]$ sudo dnf -y  install autoconf213
依赖关系解决。
================================================================================
 软件包                架构        版本                       仓库           大小
================================================================================
安装:
 autoconf213         noarch    2.13-32.fc21.loongson    fedora         167 k

事务概要
================================================================================
安装  1 Package

总下载: 167 k
安装大小: 639 k
下载软件包:
autoconf213-2.13-32.fc21.loongson.noarch.rpm
            798 kB/s | 167 kB     00:00
运行事务检查
事务检查成功。
运行事务测试
事务测试成功。
运行事务
  安装: autoconf213-2.13-32.fc21.loongson.noarch
                                         1/1
  验证: autoconf213-2.13-32.fc21.loongson.noarch
                                         1/1

已安装:
  autoconf213.noarch 2.13-32.fc21.loongson

完毕!
[loongson@localhost src]$ autoconf
../../build/autoconf/acwinpaths.m4:10: error: defn: undefined macro: AC_OUTPUT_FILES
../../build/autoconf/acwinpaths.m4:10: the top level
autom4te: /usr/bin/m4 failed with exit status: 1
[loongson@localhost src]$ cd build_debug/
[loongson@localhost build_debug]$ ../configure
Reexecuting in the virtualenv
checking for a shell... /usr/bin/sh
checking for host system type... mips64el-unknown-linux-gnu
checking for target system type... mips64el-unknown-linux-gnu
checking for the Android toolchain directory... not found
checking whether cross compiling... no
checking for pkg_config... /usr/bin/pkg-config
checking for pkg-config version... 0.28
...
Creating config.status
Reticulating splines...
Finished reading 39 moz.build files in 0.28s
Processed into 200 build config descriptors in 0.69s
RecursiveMake backend executed in 0.49s
```

```
  142 total backend files; 142 created; 0 updated; 0 unchanged; 0 deleted; 8 -> 35
      Makefile
FasterMake backend executed in 0.00s
  4 total backend files; 4 created; 0 updated; 0 unchanged; 0 deleted
Total wall time: 1.64s; CPU time: 1.63s; Efficiency: 99%; Untracked: 0.18s
[loongson@localhost build_debug]$ ls
all-tests.pkl                    binaries.json     dist      mfbt                root-deps
      .mk        test-installs.pkl
backend.FasterMakeBackend        _build_manifests  faster    modules             root.mk
                  _tests
backend.FasterMakeBackend.in     config            ipc       mozglue
      skip_subconfigures  _virtualenv
backend.mk                       config.cache      js        mozinfo.json
      subconfigures
backend.RecursiveMakeBackend     config.log        Makefile  old-configure.vars
      taskcluster
backend.RecursiveMakeBackend.in  config.status     memory    python              test-
      defaults.pkl
[loongson@localhost build_debug]$ make
[loongson@localhost src]$ cd build_release/
[loongson@localhost build_release]$ ../configure --enable-release


```

[王瑞凯]

- 时间：2019.1.1
- 机器：3421
- 地址：222.195.92.204:15901
- 执行命令 (debug)： `mkdir build_debug cd build_debug ../configure make`
- 执行命令 (release)：`mkdir build_release cd build_release ../configure --enable-release make`
- 基础测试结果 (debug)：

```
    [loongson@localhost src]$ jit-test/jit\_test.py build_debug/dist/bin/js
    [5771|   0|   0|   0] 100% ========================================>| 296.3s
    PASSED ALL

    [loongson@localhost octane]$ ../build_debug/dist/bin/js run.js
    Richards: 3682
    DeltaBlue: 7114
    Crypto: 3290
    RayTrace: 9332
    EarleyBoyer: 2518
    RegExp: 408
    Splay: 1067
    SplayLatency: 1202
    NavierStokes: 5458
    PdfJS: 1095
    Mandreel: 1747
    MandreelLatency: 2711
    Gameboy: 5103
    CodeLoad: 1671
    Box2D: 2433
```

```
    zlib: 7751
    Typescript: 3030
    ----
    Score (version 9): 2639

```

- 基础测试结果 (release):

```
    [loongson@localhost src]$ jit-test/jit_test.py build_release/dist/bin/js
    [5771|   0|   0|   0] 100% ========================================>| 279.4s
    PASSED ALL

    [loongson@localhost octane]$ ../build_release/dist/bin/js run.js
    Richards: 3650
    DeltaBlue: 5065
    Crypto: 2585
    RayTrace: 11214
    EarleyBoyer: 2601
    RegExp: 508
    Splay: 1438
    SplayLatency: 1897
    NavierStokes: 5342
    PdfJS: 1400
    Mandreel: 1958
    MandreelLatency: 2818
    Gameboy: 4051
    CodeLoad: 2685
    Box2D: 3000
    zlib: 7917
    Typescript: 3079
    ----
    Score (version 9): 2877

```

4.3 版本切换和分支创建

[王瑞凯]

- 时间：2019.1.1
- 机器：3421
- 地址：222.195.92.204:15901
- 执行命令： `git checkout firefox-52.3.0`
  `git checkout -b firefox-52.3.0-work`

5 benchmark

5.1 octane 下程序的作用

[张恺奕]

octane 是开源的 JavaScript 基准测试套件，2.0 版本包含 17 个测试文件，在 loongson 的 octane 目录下没有 SplayLatency 和 MandreelLatency 这两个测试文件。

英文参考链接https://developers.google.com/octane/benchmark?nav=true

Richards

- 操作系统内核模拟
- 主要关注点：属性的存取，函数或或方法的调用
- 次要关注点：代码的优化，冗余代码的消除

Deltablue

- 单向约束求解器
- 主要关注点：多态
- 次要关注点：OOP（面向对象编程）

Raytrace

- 光线追踪（计算机图形学的特殊渲染算法）
- 主要关注点：参数对象，应用
- 次要关注点：原型（prototype）库对象，创建模式

Regexp

- 从 50 个最受欢迎的网页抽取的正则表达式
- 主要关注点：正则表达式

NavierStokes

- 2D NavierStokes 方程求解器，要操作很多双精度数组
- 主要关注点：读和写数字矩阵
- 次要关注点：浮点数

Crypto

- 加密与解密
- 主要关注点：位操作

Splay

- 使用自动内存管理子系统，处理伸展树
- 主要关注点：对象快速创建与销毁

SplayLatency

- 插入大量检查点来测试 Splay 的 GC（垃圾回收）延迟，依延迟时间分类后，对于延迟长的打分低，延迟短的打分高
- 主要关注点：垃圾回收延迟

EarleyBoyer

- 经典的 benchmark 方案
- 主要关注点：对象快速创建与销毁
- 次要关注点：闭包，参数对象

pdf.js

- Mozilla 的 pdf 阅读器的实现，测量解码和解释时间
- 主要关注点：数组与类型数组的操作
- 次要关注点：数学运算，位运算，未来语言特性的支持

Mandreel

- 运行 3D Bullet 物理引擎
- 主要关注点：模拟

MandreelLatency

- 类似 SplayLatency 处理
- 主要关注点：编译延迟

GB Emulator

- 模拟便携式控制台的架构，以及运行所需的 3D 模拟
- 主要关注点：模拟

Code loading

- 测试 JS 引擎在加载了一段大型的 JS 程序后开始解码的速度，测试的源代码来自开源代码库（Closure, jQuery）
- 主要关注点：JS 的解析和编译

Box2DWeb

- 流行的 2D 物理引擎
- 主要关注点：浮点数运算
- 次要关注点：含浮点，访问器属性的属性

zlib

- 来自 Mozilla Emscripten 套件的 zlib asm.js/Emscripten 测试
- 主要关注点：代码编译和执行

Typescript

- 微软 TypeScript 编译自己的耗时
- 主要关注点：运行复杂，繁重的桌面应用

5.2 run.js 脚本的解释

[**张恺奕**]

run.js 包含了 octane 下所有测试文件的运行，可以通过命令../build_debug/dist/bin/js run.js 来运行

代码的注释

- load 指令首先加载 base.js，然后加载各种测试的.js 文件，加载哪些就意味着测哪些
- 变量 success 以判断测试是否成功
  - 成功-打印结果和分数
  - 失败-打印结果和错误
- base.js 定义了函数 BenchmarkSuite，其中的 config 成员定义了全局 benchsuite 运行模式，undefined 表明可用户自定义，Runsuites 成员实现了运行各种 Print 函数

5.3 jit_test.py 测试脚本

[**黄奕桐**]

- path: `src/jit-test/jit_test.py`

用途

- 基本上就是一个使用 js shell 来运行 js 脚本的测试脚本，使用该脚本能比较方便的测试 js 引擎的正确性

用法

- `jit_test.py [options] JS_SHELL [TESTS]`
  - 基本用法：`./jit_test.py ../build_debug/dist/bin/js`
- `JS_SHELL`：js shell 程序的路径
- `[TESTS]`：需要测试的脚本
  - 脚本放置在 `js/src/jit-test/tests` 目录下面
  - 新增的脚本也可以放置在上述目录下，大多放置在 `tests/basic` 下面
  - 测试特定脚本，如 `tests/test.js`(没有该文件，只是例子)
    - `./jit_test.py ../build_debug/dist/bin/js test.js`
  - 测试 tests 目录下面的 basic 目录里面所有文件：
    - `./jit_test.py ../build_debug/dist/bin/js basic`
  - 测试脚本的第一行可以添加特定的选项指定脚本的运行方式
    - 基本用法：`// |jit-test| items; [items; ...]`
    - 选项及意义：

```
slow          Test runs slowly. Do not run if the --no-slow option is given.
allow-oom     If the test runs out of memory, it counts as passing.
valgrind      Run test under valgrind.
tz-pacific    Always run test with the Pacific time zone (TZ=PST8PDT).
error         The test should be considered to pass iff it throws the given JS
        exception.
exitstatus    The test should exit with the given status value (an integer).
debug         Run js with -d, whether --jitflags says to or not
--SWITCH      Pass --SWITCH through to js
```

- `[options]`
  - 全部的选项可以使用`./jit_test.py -h` 查看，这里只列出部分
  - `-s`：show js shell command run，显示出具体运行的 js 命令

  - `-o`：show output from js shell，显示 js 输出
  - `--slow`：also run tests marked as slow，运行被标记为 slow 的 js 脚本，默认是不运行的
  - `--args=SHELL_ARGS`：extra args to pass to the JS shell，传递给 js shell 的额外参数
  - `--debugger=DEBUGGER`：Run a single test under the specified debugger，可使用`-g` 选项直接使用 gdb 调试
  - `--jitflags=JITFLAGS`：IonMonkey option combinations. One of debug, ion, all, interp, none.
  - `-R FILE`：Retest using test list file [FILE]，第一次运行时运行所有例子并将失败例子的 js 脚本名写进文件 `FILE` 里面，第二次运行时只运行文件 FILE 里面失败的例子并将再次失败的例子写进文件里面，调试时可以多次使用来测试失败的例子。
- 在运行测试例子时包含了 (include) 文件 `js/src/jit-test/lib/prologue.js`，并添加了三个常量 (global variable)：`platform`、`libdir`、`scriptdir`
  - `platform` 值为 `linux2`, 没啥用
  - `libdir` 值为`/home/loongson/mozilla-esr52/js/src/jit-test/lib/`，当要添加 lib 目录下面的某一文件时 (js 测试脚本中)，可以使用 `load(libdir + 'foo.js')`
  - `scriptdir` 值为当前测试脚本的路径
- 输出
  - 运行时会显示 `[#tests passed, #tests failed, #tests run]`
  - 如果全部运行通过，显示 `PASSED ALL`
  - 可以随时使用 `Ctrl+C` 中断测试，会显示部分结果
- 该脚本实际运行的命令
  - 运行`./jit_test.py ../build_debug/dist/bin/js -s bug828119.js`
  - 输出

```
1 /home/loongson/mozilla-esr52/js/src/build_debug/dist/bin/js -f /home/loongson/mozilla-
      esr52/js/src/jit-test/lib/prologue.js --js-cache /home/loongson/mozilla-esr52/js/
      src/jit-test/.js-cache -e 'const␣platform='"'"'linux2'"'"';␣const␣libdir='"'"'/home
      /loongson/mozilla-esr52/js/src/jit-test/lib/'"'"';␣const␣scriptdir='"'"'/home/
      loongson/mozilla-esr52/js/src/jit-test/tests/'"'"'' -f /home/loongson/mozilla-esr52
      /js/src/jit-test/tests/bug828119.js
2 PASSED ALL
```

6 perf

6.1 perf 的命令与使用

[**冉礼豪**]

- perf list
  使用 perf list 命令可以列出所有能够触发 perf 采样点的事件 (性能事件)
  格式如下:
  $perf \quad list$
- perf stat
  使用 perf stat 运行待测定的可执行文件

格式如下:

$perf \quad stat \quad ./example$

待程序运行完毕后, 将输出所记录的各个性能事件数量.

可以使用 -e 参数改变 stat 默认统计的性能事件

格式:

$perf \quad stat \quad -e \quad [event] \quad ./example$

- perf top

  格式:

  $perf \quad top$

  Perf top 用于实时显示当前系统的性能统计信息。该命令主要用来观察整个系统当前的状态，比如可以通过查看该命令的输出来查看当前系统最耗时的内核函数或某个用户进程。

  注: 也可以使用参数"-e + 事件" 来显示当前所有进程造成该性能事件的情况

- perf record/report

  使用 perf record 记录单个函数级别的统计信息，并使用 perf report 来显示统计结果。

  格式:

  $perf \quad record \quad -e \quad [event] \quad ./example$

  $perf \quad report$

  注: 使用 perf record 运行程序后会生成 perf.data 文件, 使用 perf report 查看记录的信息

  可以加"-g" 参数用来输出具体每个函数引起性能事件情况, 从而可以更加具体的判别那个函数更加需要优化

  例如:

  $perf \quad record \quad -e \quad cpu-cycles \quad -g \quad ./t1 \; perf \quad report$

- perf timechart

  这是进一步的对任务调度进行图形化的分析，可以生成矢量图, 来更加直观地观察程序运行的情况.

  格式:

  $perf \quad timechart \quad record \quad ./example//$

  $perf \quad timechart//$

  首先用 perf timechart record 记录一段时间的调度数据，再用 perf timechart 命令生成 svg 矢量图，用矢量图工具 inkscape 或者直接打开图片即可查看任务调度情况。

7 js structure

[张恺奕][黄奕桐]

7.1 JS 引擎基本结构

- SpiderMonkey 是以C和C++语言编写，并包含解释器、IonMonkey即时编译和垃圾回收器

7.1.1 基本解析流程

- 基于ECMAScript规则解析 JavaScript 代码
- 动态解析 JS 的过程

- 语法检查阶段-词法分析和语法分析
- 运行阶段-预解析和运行阶段
- JS 引擎具体解析流程
- JS 引擎首先将 JS 源代码解析成抽象语法树 (AST)
- 解释器 (interpreter) 依据语法树 AST 解释成字节码 (bytecode)，此时 JS 引擎开始执行代码
- 为了加快速度，字节码和分析数据 (profiling data) 可送到优化编译器处，优化编译器根据分析数据决定是否优化代码
- **值得注意的是：优化编译有时会出错，此时会回退到 bytecode**
- 各个 JS 引擎的区别主要是其中的 Interpreter/compiler 结构

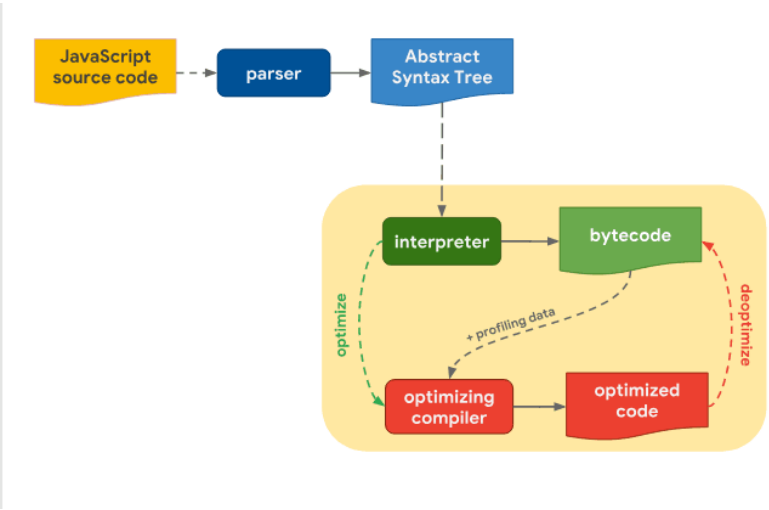


图 1: JS engine pipeline

- chrome V8 的 Interpreter/compiler 结构
- V8 的解释器名为 Ignition，其解释并执行代码，搜集 profiling data
- 优化编译器为 TurboFan，其优化代码
- firefox SpiderMonkey 的 Interpreter/compiler 结构
- Baseline会稍微 (somewhat) 优化代码，并为 IonMonkey 提供信息。**Baseline 的编译并不会出错**。
- IonMonkey 是优化编译器，能产生高度优化 (heavily-optimizied) 的代码
- JS 引擎的 Interpreter/compiler 结构主要区别是其中的优化编译器的数量，其考量依据于优化时间的开销和优化带来的收益之间的平衡 (trade-offs)

7.1.2 SpiderMonkey JSAPI blog

- JS 引擎 API 大致包括：数据类型操作、RunTime 控制、类与对象的创建和维护、函数与脚本执行、字符串操作、错误处理、安全控制、Debug 支持
- `JS RunTime` 是内存空间，包含应用程序的变量、对象和上下文 `JS Context`
- JS 引擎中一个上下文代表一个脚本，引擎传递上下文信息给运行脚本的线程
- JS API 的使用说明
- 源码 `js/src` 部分文件的作用介绍，见File_walkthrough

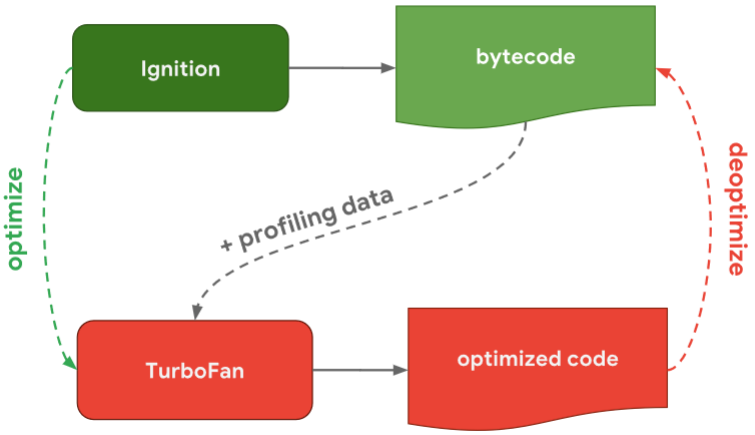


图 2: v8 interpreter/optimizing compiler

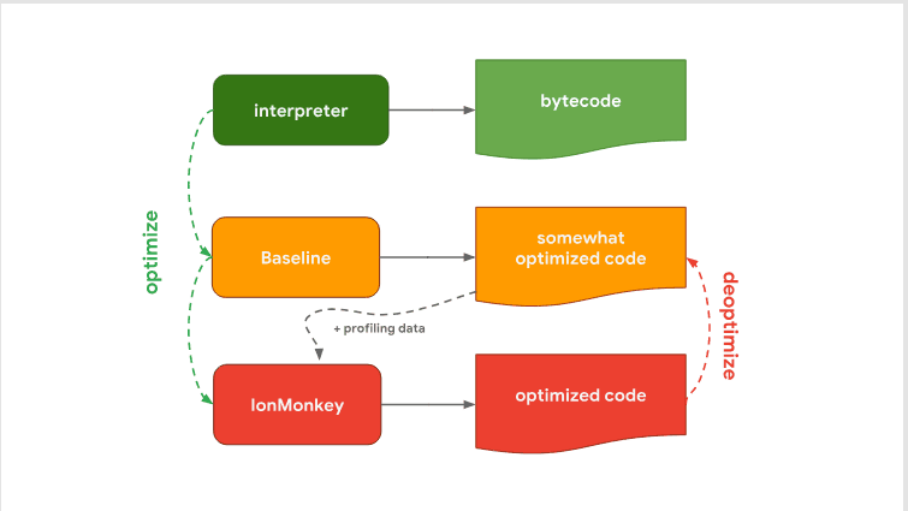


图 3: spidermonkey interpreter/optimizing compiler

JavaScript engine fundamentals: optimizing prototypes zh-CN

- 管线的优化层级-不优化启动时间早，执行时间长；优化启动时间晚，执行时间短
- 生成优化机器码需要花费很长的时间和消耗更多内存（因为字节码远短于机器码）
- Prototypes

  当谈到继承时，JavaScript 只有一种结构：对象。每个实例对象（object ）都有一个私有属性（称之为 [[prototype]]）指向它的原型对象（prototype）。该原型对象也有一个自己的原型对象 ，层层向上直到一个对象的原型对象为 `null`。根据定义，`null` 没有原型，并作为这个原型链中的最后一个环节

- 原型属性的访问过程
- 有效性验证单元（解决原型属性的从下往上访问导致大量的时间开销）

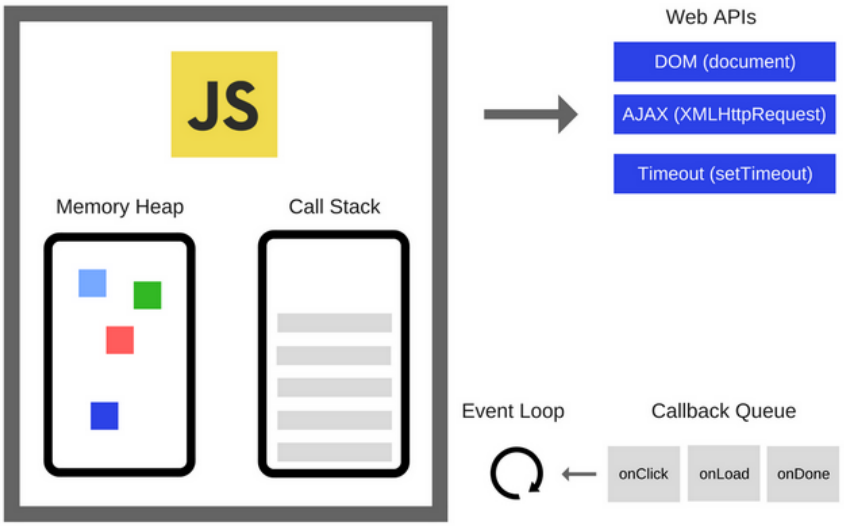


图 4: JS engine runtime

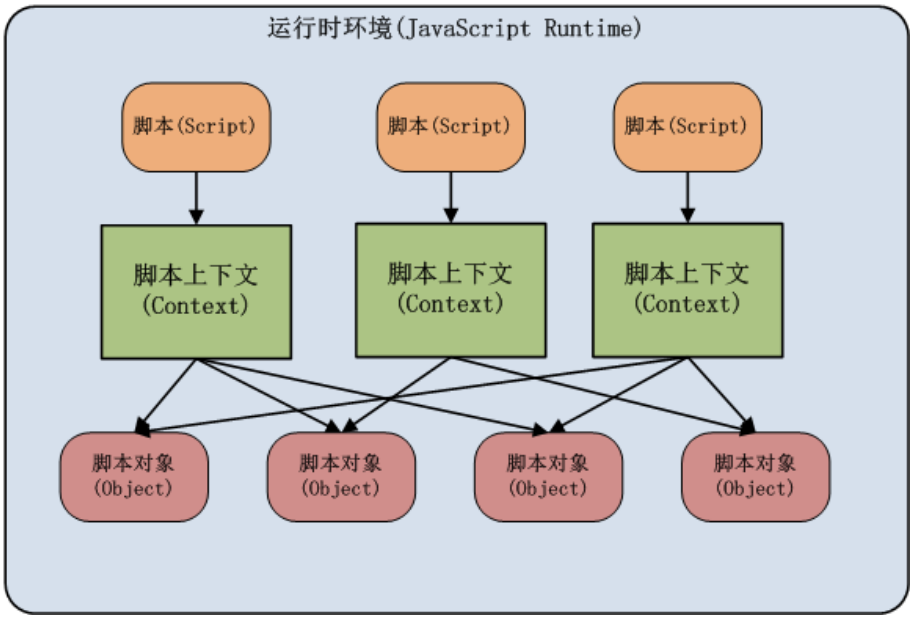


图 5: JS runtime

- 思想就是利用 Inline Cache 对原型中的属性进行直接访问
- Inline Cache 被命中生效时，JS 引擎检查实例的 Shape 以及其 ValidityCell
- ValidityCell =1，就可以直接访问原型的的属性，不用从下往上遍历查找

7.2 Shape and Inline cache

JavaScript engine fundamentals: Shapes and Inline Caches zh-CN

- Shapes 和 Inline Caches 目的是减少 object 存储空间，加速访问 JS 对象的属性
- JS 的 objects 对象存储为字典结构 (dictionary)，其 key 以字符串的形式存储，每个 key 有相应的 property attributes

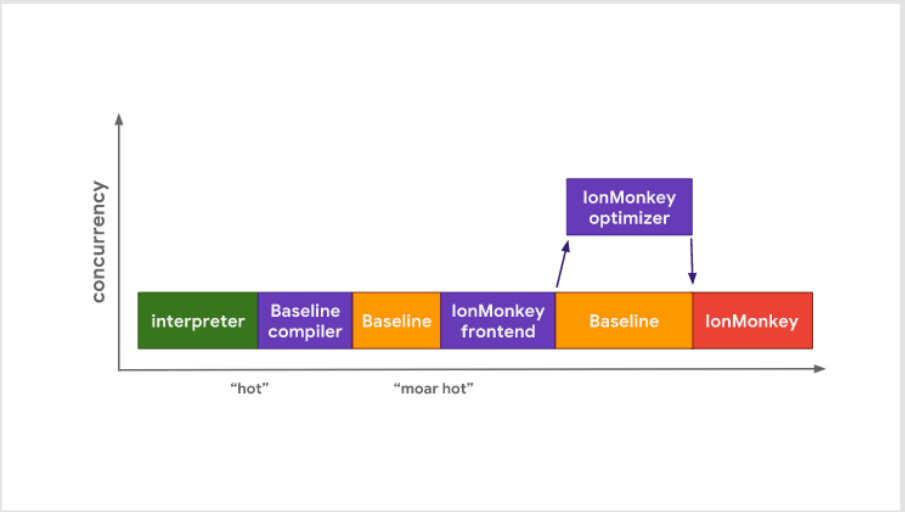


图 6: 管线

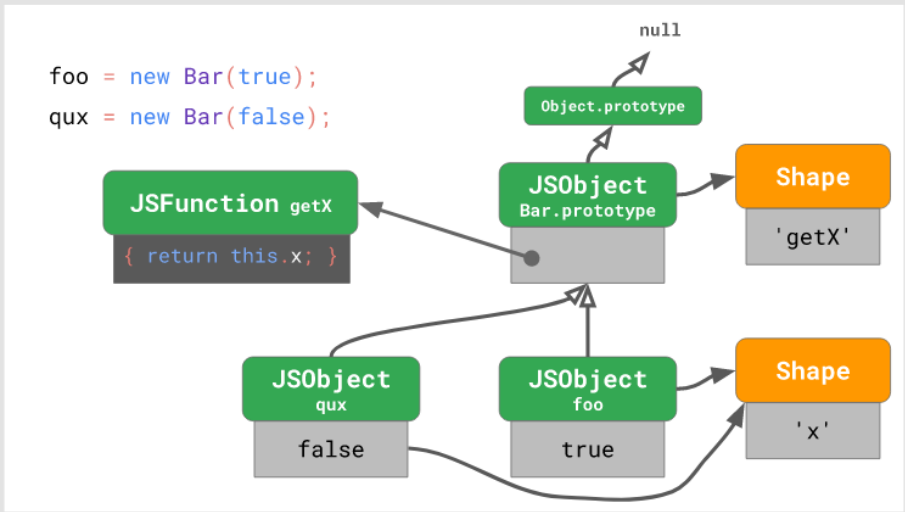


图 7: Prototypes

- JS 编程中，给不同对象起相同的 key 名很常见，访问不同对象的同一个 key 很常见
- Shapes
  - 该结构便于后面的 inline cache 实现缓存机制
  - 拥有 object 的共性，这样每个 object 不必有 key，节省空间
- 增加属性的时候，以一种 transition chains/trees 的方式实现
- 属性太多怎么办，获得属性就会很慢，使用 ShapeTable
- 数组虽然也是 object，但是对下标做了特殊的优化，没有纳入 shape
- Inline Caches
  - 局部缓存，JS 引擎为了提高对象查找效率，需要在局部做高效缓存
  - 把 shape 缓存下来是不是加快查找速率，是的!!!
  - 可以更快，Shape 的 offset 可以缓存起来，下次开始直接跳过 Shape 这一步

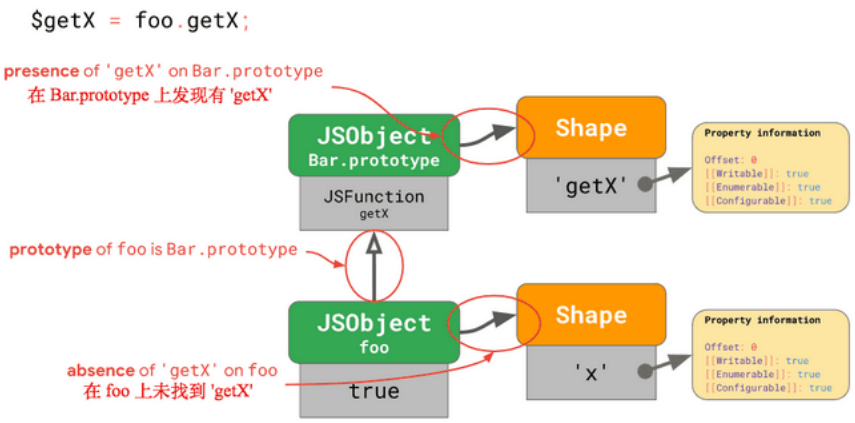


图 8: 原型属性的访问过程

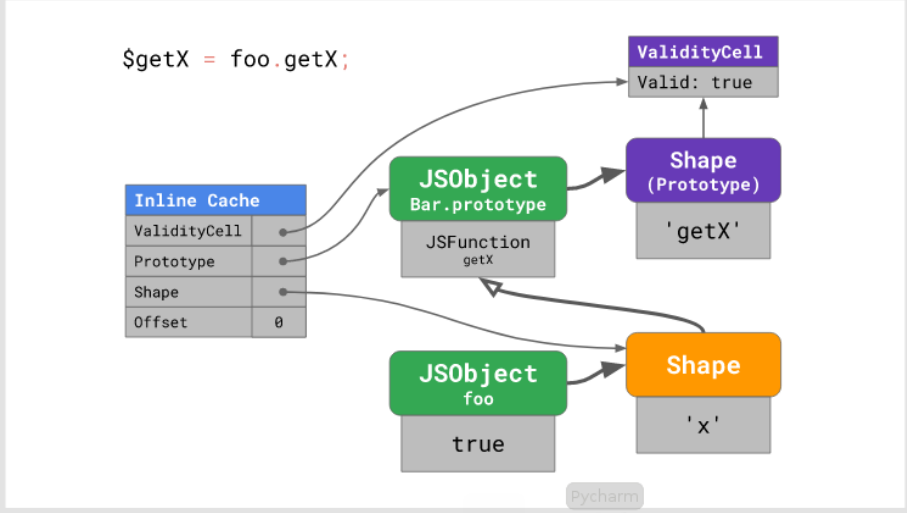


图 9: 有效性验证单元

JavaScript engine fundamentals: optimizing prototypes zh-CN

- 管线的优化层级-不优化启动时间早，执行时间长；优化启动时间晚，执行时间短
- 生成优化机器码需要花费很长的时间和消耗更多内存（因为字节码远短于机器码）
- Prototypes

  当谈到继承时，JavaScript 只有一种结构：对象。每个实例对象（object ）都有一个私有属性（称之为 [[prototype]]）指向它的原型对象（prototype）。该原型对象也有一个自己的原型对象 ，层层向上直到一个对象的原型对象为 `null`。根据定义，`null` 没有原型，并作为这个原型链中的最后一个环节

- 原型属性的访问过程
- 有效性验证单元（解决原型属性的从下往上访问导致大量的时间开销）
- 思想就是利用 Inline Cache 对原型中的属性进行直接访问
- Inline Cache 被命中生效时，JS 引擎检查实例的 Shape 以及其 ValidityCell

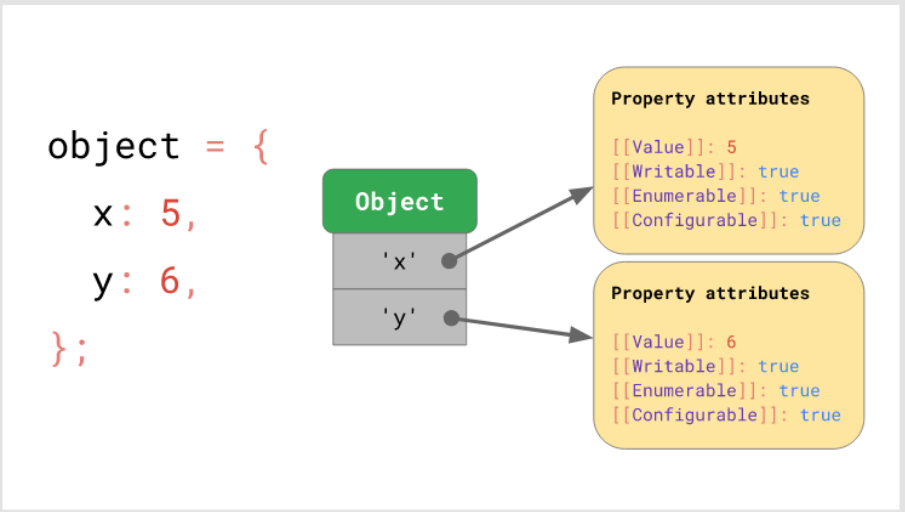


图 10: objects' property attributes

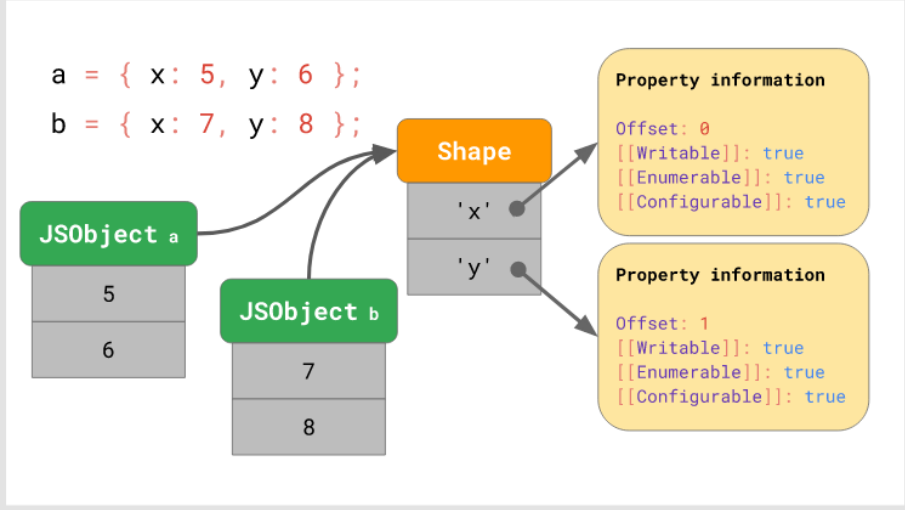


图 11: use shape

- ValidityCell =1，就可以直接访问原型的的属性，不用从下往上遍历查找

7.3 Shape and Inline cache source

7.3.1 ref

Javascript Hidden Classes and Inline Caching in V8 Optimizing dynamic JavaScript with inline caches

Hidden Classes/V8

the purpose of the hidden classes is to optimize property access time same as shape/transition chains and trees in spidermonkey 以下的例子说明，对象属性的创建顺序不同就会出现不同的 hidden classs，不然的话，hidden classes 就可以由不同的 object 共用

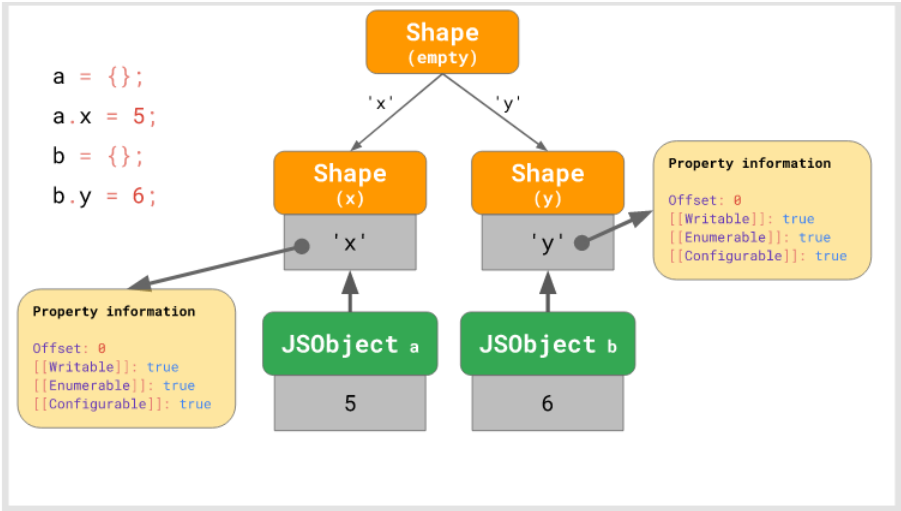


图 12: use shape chains

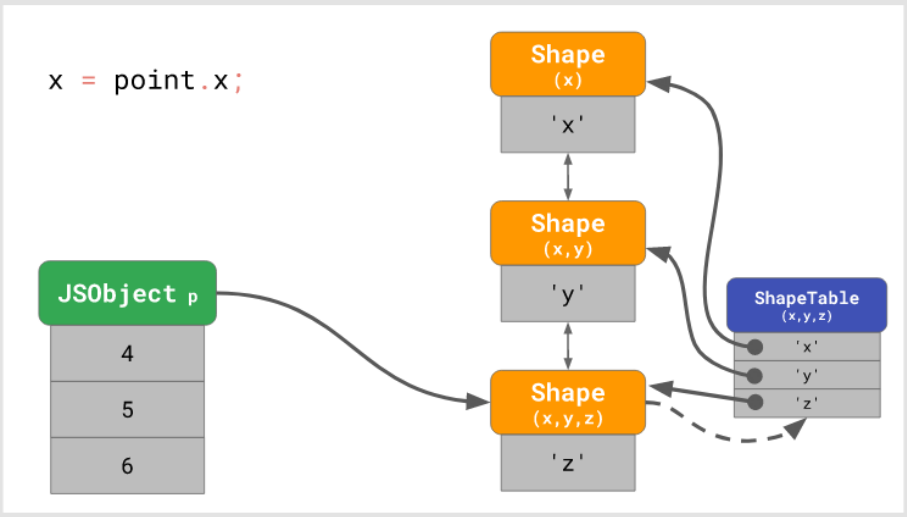


图 13: use shapetable

```
1  function Point(x,y) {
2    this.x = x;
3    this.y = y;
4  }
5
6  var obj1 = new Point(1,2);
7  var obj2 = new Point(3,4);
8
9  obj1.a = 5;
10 obj1.b = 10;
11
12 obj2.b = 10;
13 obj2.a = 5;
```

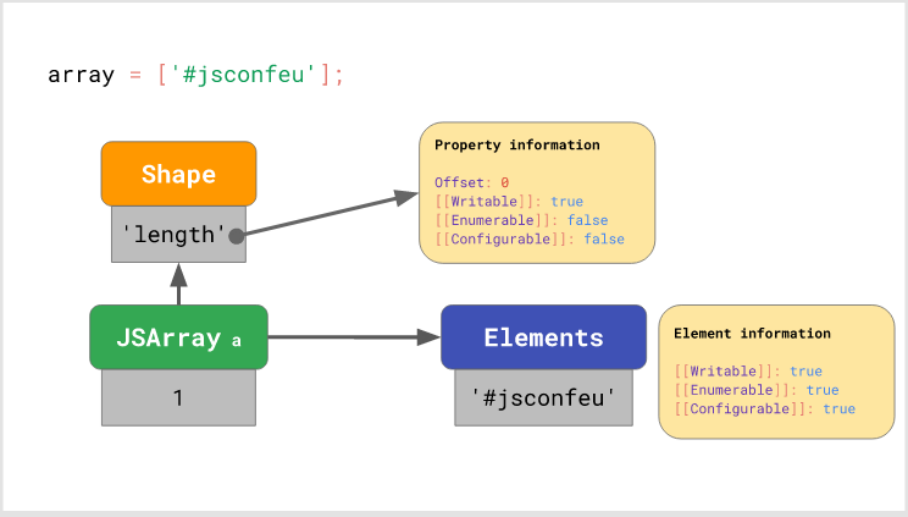


图 14: optimization in array

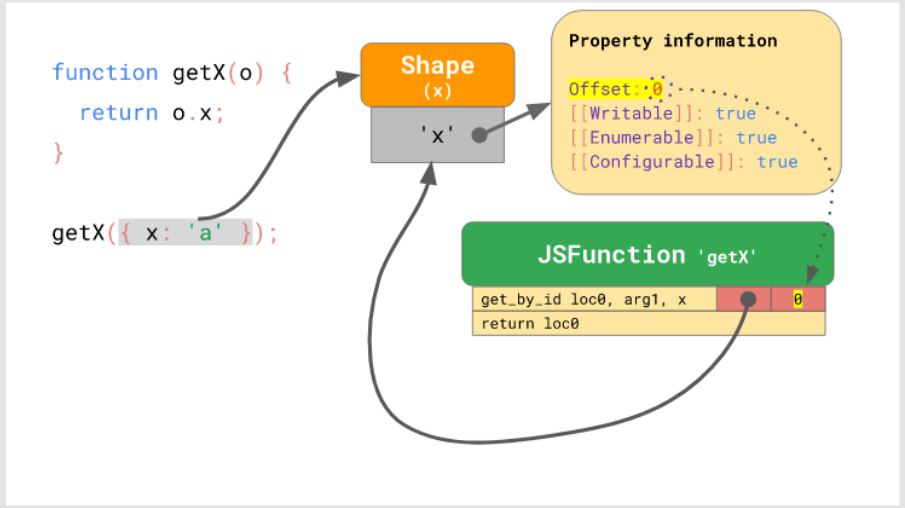


图 15: use inline cache

7.3.2 Inline Caching/V8

inline caching relies upon the observation that repeated calls to the same method tend to occur on the same type of object. 结合 inline caching 和 hidden classes

1. lookup to that objects hidden class to `determine the offset` for accessing a specific property// 获得具体属性的 offset
2. After two successful calls of the same method to the same hidden class, V8 omits the `hidden class lookup` and simply adds the offset of the property to the `object pointer` itself// 经常访问，对象指针自身缓存 offset
3. For all future calls of that method, the V8 engine *assumes* that the hidden class hasn't changed, and jumps directly into the memory address for a specific property using the offsets stored from previous lookups// 直接由指针的 offset 访问对应的内存位置

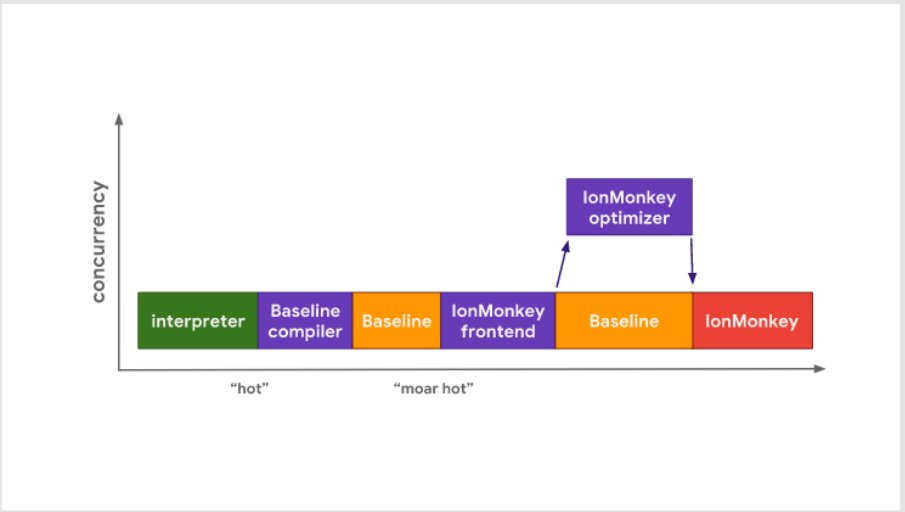


图 16: 管线的优化层级

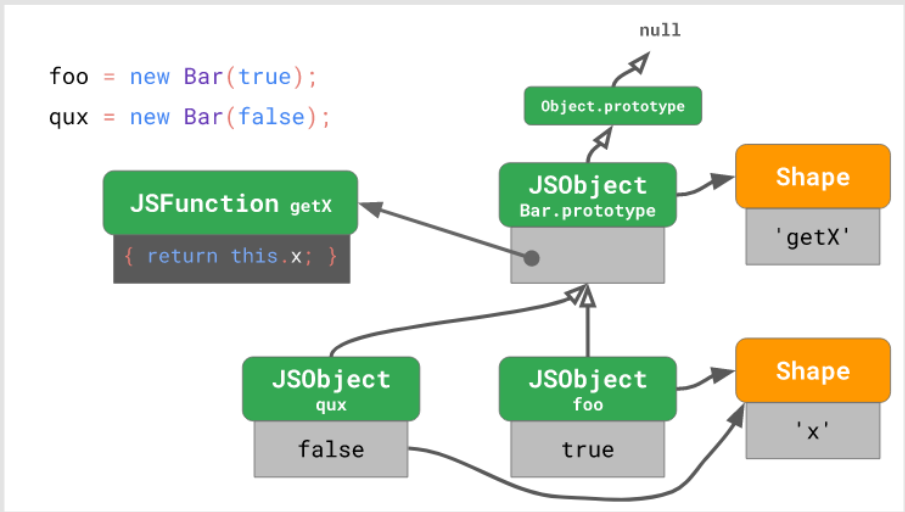


图 17: 原型 prototypes

7.3.3 Shape Source/SpiderMonkey

源码位于 js/src/vm/Shape.h js/src/vm/Shape.cpp

shape_->shape pointer

- object pointer 和 shape pointer 缓存 offset 代码实现在 jc/src/jit/SharedIC.h 的 ICGetPropNativePrototypeStub 类
- holder() 返回未实例化的 object pointer，offsetOfHolderShape() 实现缓存 object pointer 指向的 shape 链的属性的 offset

shape lineages

- 一个 shape lineages 代表一个具体的 object
- shape lineages 的组织形式
  - N-ary property trees(共享 shape)
  - Dictionary mode lists(不共享 shape)

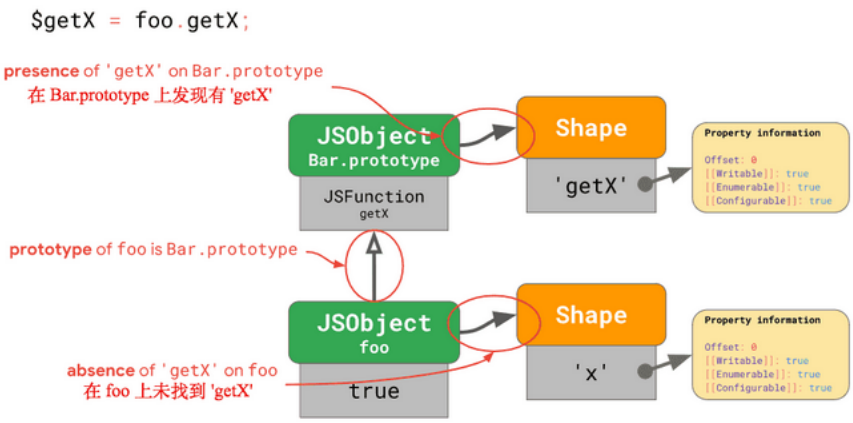


图 18: 原型属性的访问过程

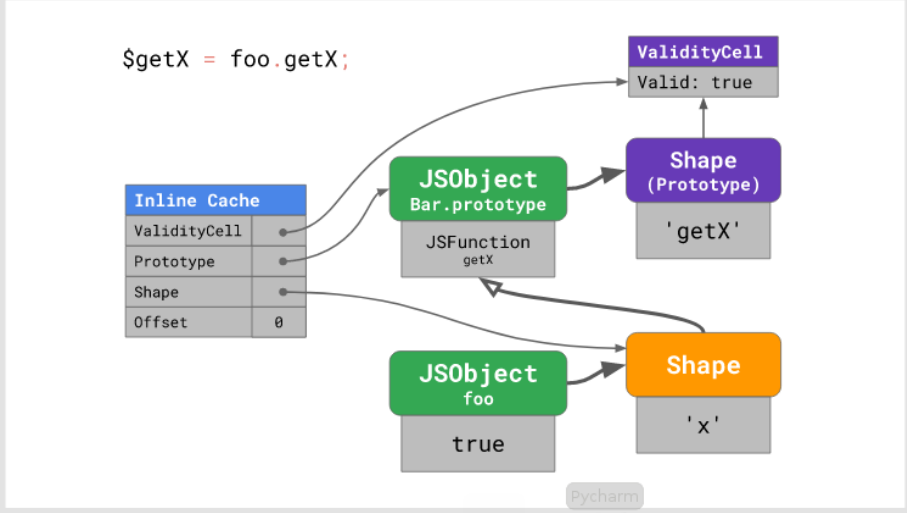


图 19: 有效性验证单元

- shape lineages 的 link 都是双向的
- N-ary property trees 转换为 Dictionary mode lists 的条件
  - shape lineage's size reaches MAX_HEIGHT
  - A property represented by a non-last Shape in a shape lineage is removed from an object.
  - A property represented by a non-last Shape in a shape lineage has its attributes modified.
- hash table – the ShapeTable
  - 有 point_ 指向处开始搜索某个属性，搜索深度大于 LINEAR_SEARCHES_MAX 还未搜到，就将该 shape 直接加入 shape table，就可以在 O(1) 时间内找到
  - 可被 GC 回收
  - AutoKeepShapeTables 不可被 GC 回收

BaseShape

- shape 的 shape，很多 shape 有相同数据，可以抽象形成 baseshape
- shape->base() 可以将 shape 的属性的编码信息划分到 shape 和 baseshape

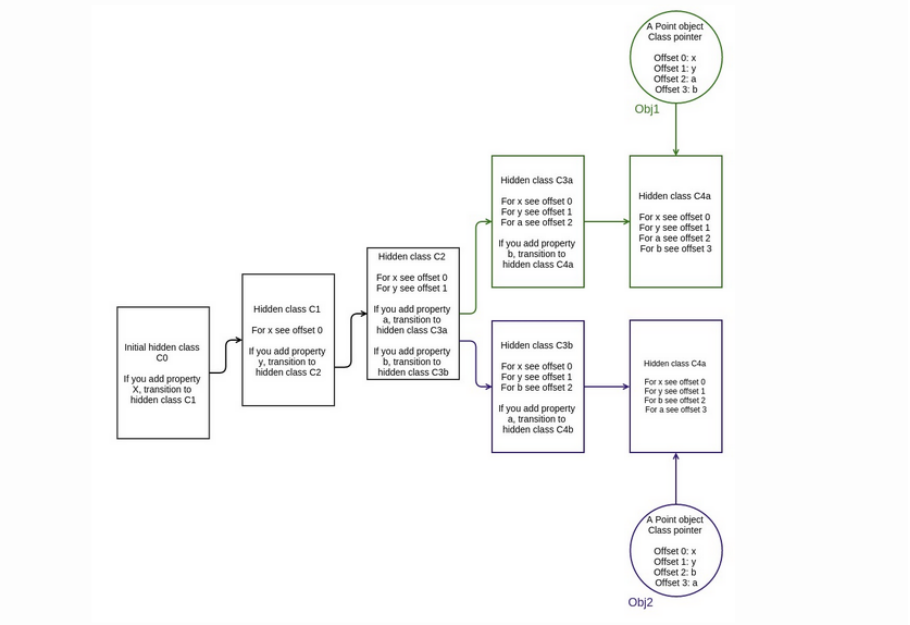


图 20: V8 hidden class example

```
2452  class ICGetPropNativePrototypeStub : public ICGetPropNativeStub
2453  {
2454      // Holder and its shape.
2455      GCPtrObject holder_;
2456      GCPtrShape holderShape_;
2457
2458    protected:
2459      ICGetPropNativePrototypeStub(ICStub::Kind kind, JitCode* stubCode, ICStub* firstMonitorStub,
2460                                   ReceiverGuard guard, uint32_t offset, JSObject* holder,
2461                                   Shape* holderShape);
2462
2463    public:
2464      GCPtrObject& holder() {
2465          return holder_;
2466      }
2467      GCPtrShape& holderShape() {
2468          return holderShape_;
2469      }
2470      static size_t offsetOfHolder() {
2471          return offsetof(ICGetPropNativePrototypeStub, holder_);
2472      }
2473      static size_t offsetOfHolderShape() {
2474          return offsetof(ICGetPropNativePrototypeStub, holderShape_);
2475      }
2476  };
```

图 21: ICGetPropNa-tivePrototypeStub 类

gcptr

- 源码位于 js/src/gc/Barrier.h
- GCPtr must be managed with GC lifetimes
- class shape 里有两种实现
  - GCPtrShape，T=Shape
  - GCPtrBaseShape，T=baseshape

Owned/Unowned

- Owned Shape, Owned BaseShape: 用于以 Dictionary mode lists 组织的 shapes 的属性，包括 last property
- Owned Shape, Unowned BaseShape: 用于以 Dictionary mode lists 组织的 shapes 的属性，不包

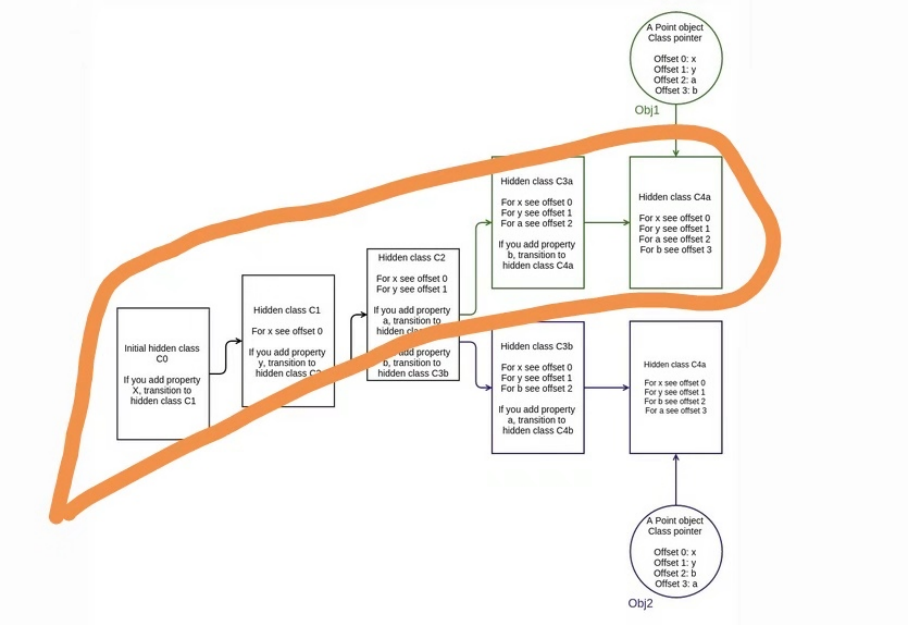


图 22: shape lineages

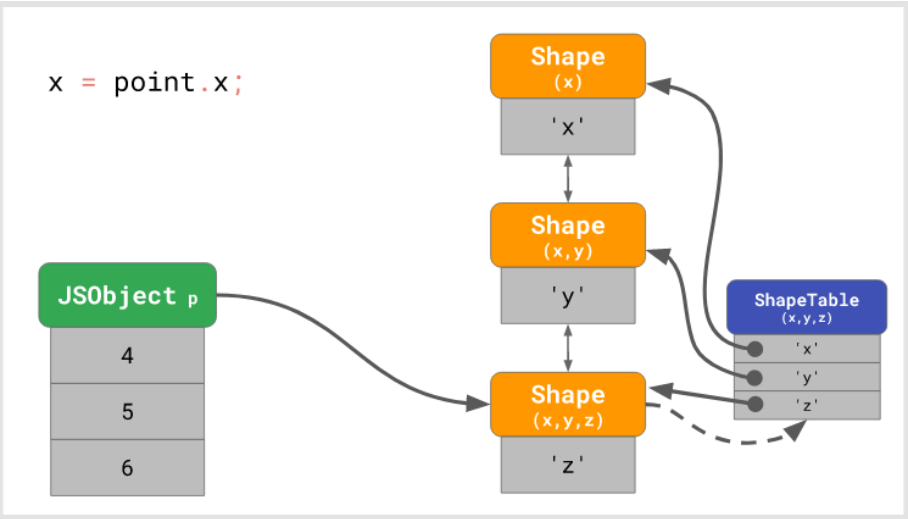


图 23: the ShapeTable

括 last property

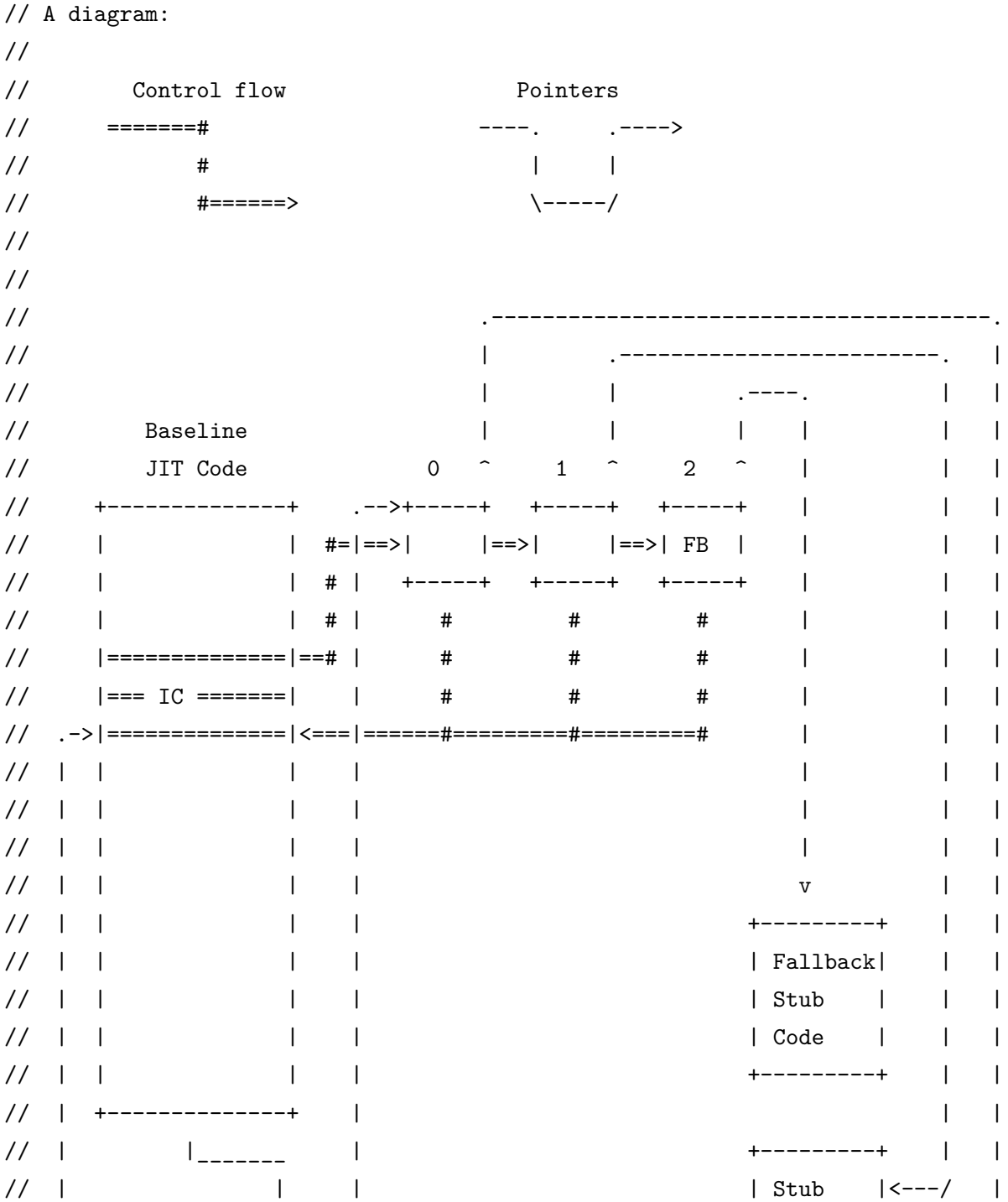
- Unowned Shape, Owned BaseShape: 用于有 shape table 的以 N-ary property trees 组织的 shapes 的属性
- Unowned Shape, Unowned BaseShape: 用于没有 shape table 的以 N-ary property trees 组织的 shapes 的属性

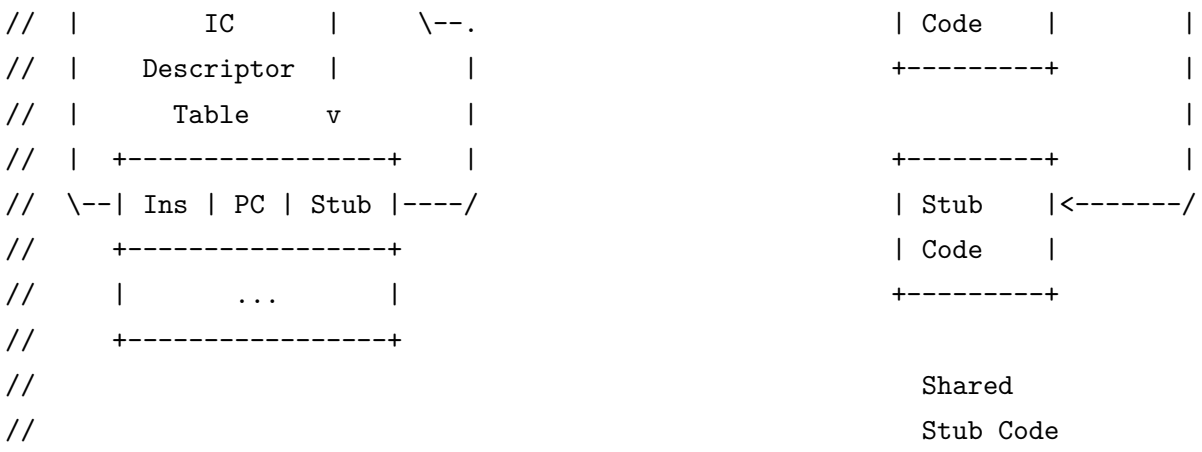
7.3.4 IC

Shared IC(js/src/jit/SharedIC.h)

- Inline Cache 是一个多态的 Cache(`polymorphic caches`), 并且和其他的 IC 共享 IC 代码 `stub code`。

  - 多态这里应该指那些运算对象可以是不同类型的，比如 . 号（取成员）的运算对象类型可以是不同类型的 object 对象
- Inline Cache 在对应的位置是以链表组织的 (`a linked list of stubs`)，每一项 (`stub`) 代表不同的类型
  - 每一项包含了与该项有关的数据，如指向 shape 的指针
- 每一个 `BaselineScript` 都有一个 `IC Descriptor Table`
  - 每一个表项包含一个指向 IC 链表头的指针，IC 相关联的字节码的 PC，还有 IC 代码的返回地址 (`machine-code PC`，这里应该是编译后的汇编地址)
- 下面是注释中的一张图，展示了 IC 的结构





8 gc

王瑞凯

8.1 Design Overview

main datastructure

Cell 一组内存 — 从 GC 外部看，GC 的工作就是给 cell 分配内存并自动收集它们 — 是所有 GC 收集的类的基类 Allocation Kind 指示分配类型，决定对象的大小和析构 (finalize) 行为。— 数据结构

enum AllocKind — Arenas8.1储存 allocation kind 相同的对象（大小相同，析构行为相同）

Compartments

JS 堆被分成各个 compartments。性质：

- 每个 cell 属于至多一个 compartment。(但一些 cells 被一个 Zone 的所有 compartment 共享)
  注 8.1 这个性质有点矛盾
  另外可以看出 *zone* 的范围比 *compartment* 大
- 一个对象不能持有指向另一个 compartment 的对象的指针，只能储存一个 wrapper Q. 啥是 wrapper? 好处是允许 compartments 做安全检查：相同 compartment 里的对象访问要求 (access erquirements) 是相同的，不需要检查。但跨 compartment 就要检查。
- 可能只 gc 一个 compartment，也可能一次 gc 多个 compartments。用跨 compartment 的 wrappers 作为根 Q. 什么意思
- Q. cross-cutting comcept

Zone

一个 zone 是一个 compartment 的集合，被用作 gc 的边界 (boundaries)。gc 收集操作在 zone 的级别进行，而不是 compartment。zone 的性质

- 每个 compartment 属于唯一的 zone
- 每个 js 堆对象属于唯一的 zone
- 在同一个 zone 但不同 compartment 的对象可以共享一个 arena
- 不同 zones 的对象不能被储存在一个 arena

- 一个 zone 是活跃的 (alive)，当这个 zone 里的任意 js 堆对象 (heap object) 都是活跃的

Chunk

- 最大的内部内存块 (unit of memory allocation)
- 1MB
- Arenas, padding, the mark bitmap, a bitmap of decommitted arenas, chunk header(ChunkInfo)
  Q. 信息量庞大
- ChunkInfo 包括一列未分配的 Arenas，从 ChunkInfo::freeArenasHead 开始，用 ArenaHeader::next 链接，也储存一些基础的状态参数 (stats)，例如空闲 arenas 的数量

Arena

- 内部内存分配块 (internal unit of nenory allocation)
- 一页 (4k)，包含 ArenaHeader，pad bytesQ. pad，一组对齐的元素 (aligned elements)。**一个 Arena 里的所有元素具有相同的分配类型和大小**
- 维护一个空闲 spans。从 ArenaHeader::firstFreeSpanOffets 开始，除最后一个空闲 span，空闲 span 的最后一个 cell 维护一个 **FreeSpan??**，指向下一个空闲 span

Free Span

数据结构：struct FreeSpan
位置：src/gc/Heap.h line 369
一组连续的空闲的 cells[first,last]，包含从空闲 span 分配 (应该是 cell) 的成员函数

Mark Bitmap

数据结构：ChunkBitmap
每个 GC cell 两位，表示 liveness marking(黑) 和 cycle-collected marking(灰)Q. cycle-collected marking

incremental marking

增量标记：做一点标记工作，然后让 mutator 跑一会儿，然后在做一些标记工作。**好处：避免了长时间的暂停，暂停可以置为 10ms 或更少**

有可能需要长时间的暂停。如果分配速度是快的，引擎可能在结束增量 GC 之前用完内存。如果这样的情况发生，引擎必须立即开始一个 full，non-incremental GC 来回收一些内存并继续执行

Incremental write barrier

基本的问题：对象 A 是黑色的，而且包含一个指针域。让 B 是白色的。现在增量时间片 (incremental slice) 停止了，所以 mutator 开始工作。如果 mutator 将 B 储存到 A 里（应该是说把 B 的引用/指针储存到 A 的指针域里），然后删除所有指向 B 的指针。那么：

- B 活着，因为 A 是黑的而且指向 B
- B 不会被标记，因为 B 只能通过 A 到达，A 是黑的
- 所以，B 活着但是会被回收

注 8.2 根本问题在于这段时间里（*A* 是黑色），*mutator* 对 *A* 的指针域进行修改，应该对被指向的对象进行标记处理。因为 *A* 不会再被访问了。

write barrier 是小段代码，在一个指针赋值 (pointer store) 行为发生前执行，记录足够的信息保证活着的对象不会被回收

The SpiderMonkey incremental write barrier

snapshot-at-the-beginning allocate-black-barrier

假设我们打算做一个增量 GC，在 GC 期间，没有分配新的对象。如何保证不会回收任何活着的对象？一个方式是保证：在增量标记开始的时候，每一个对象都活着（意味着如果指向一个对象的所有引用在这次 GC 中 drop 了，那这个对象会在下一次 GC 里被回收）— snapshot-at-the-beginning (这样的现场保留实现上 (physically) 是不需要的）— 实现上需要一次 full nonincremental 标记

保留现场的 barrier 在实现上是容易的。barrier 在 Mutator 准备重写一个 GC 指针域的时候启动。barrier 将被指向的对象标记为黑，and pushes any outgoing edges onto the mark stack(?)。关键的考虑是保证只有在指向一个对象的所有指针都被重写之后，这个对象才不被标记。所以，一旦我们打算重写指向一个对象的指针，先把这个对象标黑，保证没有对象会丢失。

注 8.3 这个逻辑就是，任何一个可能丢失的对象，先假设它是活的。即使所有的指针都被修改了，它也会在下一轮被回收。

现在考虑分配问题。一个新分配的对象没有出现在 GC 的开始，所以保留现场没有作用。但是我们需要保证这个对象在活着的时候不被回收。这也是简单的，只要在分配的时候立即将新对象标记为黑，这个策略被称为 **allocate-black**

The SpiderMonkey incremental read barrier

增量 GC 的教科书版本只有写屏障。SpiderMonkey 还需要有个读屏障。因为弱引用 (weak reference) 有一个与写屏障解决的问题类似的问题。但是需要不同的解决方案。

弱引用不应该通过自己来维护指向的对象活着。这意味着我们无法跟踪弱引用，在 GC 图遍历计数中。我们将他们挂在一个列表里。在标记了所有活对象之后，我们扫描这个列表以 (null out references to garbage)

注 8.4 这里的意思是不应该通过弱引用标记任何对象

但是这同时也意味着事先屏障 (pre-barrier) 是不足以防止增量 GC 因为图修改而错过一些便。活着的 cell 会被标记，或者因为他们是可达的，或者因为他们在被 GC 之前被实现屏障标记了。但是可达的弱引用没有被跟踪！所以你可以读一个弱引用，然后后台 (behind the frontier) 对他进行写操作，那么没有事先屏障会标记它。所以加了写屏障，如果你在增量 GC 期间读了一个弱引用，那它指向的对象就被标记了。

注 8.5 这里的逻辑在于弱引用对象的存活与否不取决于这个引用是否存在，而是程序员是否访问这个对象。那么当程序员访问这个对象的时候，增量 *GC* 就要保证这个被访问的对象要活着

implementing details 写屏障有执行开销 (runtime cost)，所以 SpiderMonkey 尝试在增量 GC cycle 不活跃的时候跳过它们。每个 Zone 有一个标志 (flag) `needsIncrementalBarrier()`，指示是否需要屏障。

对每一个类型 `T` 的某个域 (fields of type `T*`) 需要一个写屏障，用函数 `T::writeBarrierPre(old)`。例如，类型 `JSObject*` 的域需要写屏障，使用函数 `JSObject::writeBarrierPre(JSObject *old)`。如果 `zone->needsIncrementalBarrier()` 为真，则 `writeBarrierPre()` 标记 `old`。

类 `Heap<T>` 用于简化写屏障的调用。`Heap<T>` 封装了一个 `T*` 并在赋值的时候调用写屏障

注 8.6 这个数据结构本质上就是一个指针的拓展，包含一个指针，当这个指针改变时调用写屏障

类似的数据结构还有

HeapValue	Value
HeapSlot, HeapSlotArray	slots
HeapId	jsid

对象的私有域需要被特别处理。私有域本身对引擎是不可见的 (opaque)，但是它可能指向一些需要被标记的对象。例如：一个 js 对象指针数组。在这个例子中，如果私有域被重写，这些对象指针就丢掉了，所以需要调用写屏障来标记它们。`js::NativeObject::privateWriteBarrierPre` 用于处理这类情况，通过在私有域被重写之前调用 js 对象的类追踪链 (class trace hook)

另外一个细节是写屏障在初始化新对象域的时候可以被跳过，因为没有指针被覆盖。

注 8.7 这里我才真正明白屏障想要解决的问题。举一个更加具体的例子，以上面 *js* 对象指针的数组为例，如果在某一轮的标记的时候，这个数组被覆盖了，没有调用写屏障，而且这个数组指向的对象的其他引用者都还没有扫描到，那么这些对象就不会被标记，会在这一轮 *cycle* 之后被回收。那么一段时间之后，这些对象的其它引用者打算调用这些对象的时候，这些对象实际上已经不存在了。这时候程序就会出错。

source file

gc/GC.h,cpp Defines GC internal API functions, including entry points to trigger GC.

gc/Barrier[-inl].h Implements incremental and generational write barriers.

gc/Cell.h Defines Cell and TenuredCell.

gc/Heap.h Defines Chunk, ChunkInfo, ChunkBitmap, Arena, ArenaHeader, Cell, and FreeSpan structures that define the heart of the GC heap's structures.

gc/Marking.h,cpp Defines all marking functions for the various garbage-collected things.

gc/Memory.h,cpp Contains a few functions for mapping and unmapping pages, along with platform-specific implementations. The map/unmap functions are used by GC.cpp to allocate and release chunks. There are also functions to commit or decommit memory, i.e., tell the OS that certain pages are not being used and can be thrown away instead of paged to disk.

gc/Root.h Defines classes for noting variables as GC roots.

gc/Statistics.h,cpp Defines struct Statistics, which stores SpiderMonkey GC performance counters.

Dictionary of terms

黑色

这个对象被标记了，它的孩子 (children) 是灰色的。被标记 — 活着

灰色

等待标记 (queued for marking)

在 SpiderMonkey 里，灰色表示该对象不是黑色，但是可以从“灰色根结点”到达。灰色根结点的使用者为 Gecko cycle collector Q. ，功能为找到遍历 (pass through) 整个 js 堆。(在标记栈里但不是黑色)

白色

白色: 不在标记栈而且没被标记
mark phase 结束后还是白色 — 表示没被标记

注 8.8 这个地方的灰黑白应该类似于深度优先遍历的栈操作。黑色是指已经出栈的结点，已经被标记。它的孩子结点，已经入栈，但还没被标记（没出栈），所以是灰色的。而在图中不可达的结点，自始至终就没进过栈，所以是白色的。

弱指针 (pointer)

在计算机术语中，弱指针：在 GC 环境下不需要保证指向的对象活着 (one that doesn't keep the pointed-to value live for GC purposes)。特别的，如果指向的对象被回收了，那么弱指针的 get() 方法返回空指针。

在 Gecko/SpiderMonkey，弱指针指向没被标记的、可以被回收的对象。程序员必须保证被指向对象的生存周期包含弱指针的生存周期 (the lifetime of the pointed-to object contains the lifetime of the weak pointer)

注 8.9 文档这里写了一个 *TODO make sure this is correct*。

还有一些标记 TODO 的文档里没有完成的部分：

- Marking
- Sweeping
- Generational GC

8.2 jsgc.cpp

[SMDOC] 垃圾回收
代码实现了一个**增量的标记清除**垃圾回收器，大部分的回收都是在后台的并行线程中执行的

Full vs. zone GC

回收期一次回收所有的区域 (zone) 活着只回收其中的一部分。这样的两类垃圾回收策略分别被称为 full GC 和 zone GC。一个增量 GC 可能以 full GC 的形式开始，但因为在执行期间有新的区域产生，所以最终编程 zone GC

incremental collection

增量回收必须满足的条件有：

- 垃圾回收器执行 `js::GCSlice()`而不是`js::GC()`
- 回收模式必须被标记为`JSGC_MODE_INCREMENTALGC.cpp:342` with `JS_SetGCParameter()`
- 任何线程的栈不能含有`AutoKeepAtoms`实例
- 工程上的考虑：**使用合适的 barrier 机制**

非增量的垃圾回收

- 如果一次回收不是增量的, 所有的前台活动发生在一次`GC()` 或 `GCSlice()`调用中. 知道后台的清除工作结束一次 GC 才算完成。
- 一次增量 GC 分为一系列的时间片进行执行，时间片之间穿插执行 js 代码。时间片被限制在一个特定的时间预算中，当执行时间超过预算时间时，增量 GC 以尽可能快的速度停止执行

垃圾回收器状态

- MarkRoots - 标记栈和其他根结点
- Sweep - 清楚被标记的区域，继续标记未清除的区域
- Finalize - 后台执行，与 js 代码并发
- Compact - incrementally compacts by zone
- Decommit - performs background decommit and chunk removal

[1,markroots:MarkRoots 总是在第一个时间片执行，第二个和第三个状态可以在之后执行]

执行过程

Slice 1:

- MarkRoots: 将根节点加入标记栈
- Mark: 标记栈弹出结点，标记，然后压入孩子节点.
- ... JS 代码运行...

Slice 2:

- Mark: 更多的标记栈操作
- ... JS 代码运行...

Slice n-1:

- Mark: 更多的标记栈操作
- ... JS 代码运行...

Slice n:

- Mark: 标记栈被清空 (drained)
- Sweep: 选择第一组区域清除 (sweep)
- ... JS 代码运行...

Slice n+1:

- Sweep: Mark objects in unswept zones that were newly identified as alive (see below). Then sweep more zone
- Sweep: 标记未清除区域中的被重新认为 alive 的对象。然后清除更多的区域
- ... JS 代码运行...

Slice n+2:

- Sweep: 标记未清除区域中的被重新认为 alive 的对象。然后清除更多的区域
- ... JS 代码运行...

Slice m:

- Sweep: 清除结束，后台清除以一个辅助线程的方式启动
- ... JS 代码运行... ，剩余的清除工作在后台线程中完成

[2,GC complete: 当后台清除结束后，一次 GC 结束]

Incremental marking

- 增量 GC 需要与 js 代码运行精密合作来保证正确性
- 在一次增量 GC 中，如果一块内存空间（除了根）被写入内容，那么它之前保有的值必须被标记。写屏障 (write barrier) 就是用来保证该条件的。
- 在增量 GC 执行过程中创建的对象需要被标记
- 根在最开始就被标记了，所以不需要写屏障
- 根是类似 C 的栈或虚拟机栈的东西
- 使用现场保存 (snapshot-at-the-beginning) 算法来实现上述内容。这意味着我们在收集开始的时候至少所有可达的对象都会被标记 Barrier.h.

[3,barriers application: 写屏障解决的问题是在两次回收的 slice 之间 js 代码可能会改变对象图 (object graph)，我们必须保证这不能导致 GC 标记过程中漏掉一个可达的对象]

增量清除

清除是很难增量执行的因为 **对象的析构 (finalize) 必须在清除开始时就执行，在任何 js 代码运行之前**. 原因是一些对象使用他们的析构函数将他们从缓存中移出来。如果 js 代码被允许在清除开始之后执行，它可能会访问 (observe) 缓存的状态并建立一个新的引用到即将清除的对象。

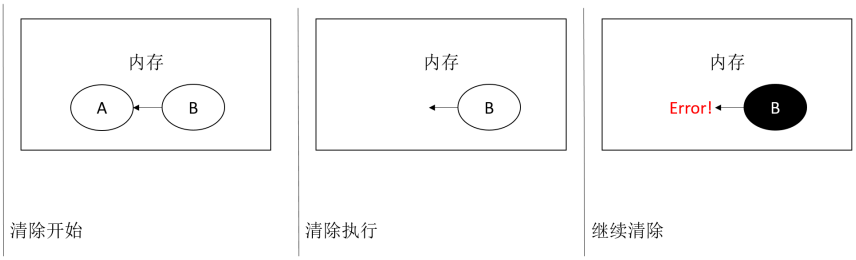


图 24: 增量的清除可能出现的问题

图24是……

- 在一次执行中清除所有的可析构的对象会导致长时间的暂停，所以清除区域被分组执行。没有被清除的组还是要被标记，所以上述的情况不会发生。
- (描述了一个具体的场景。sweep 清除掉一个对象，但 mutator 通过另外一个对象 mark 这个对象，但此时这个对象已经被清除掉了) 执行清除的顺序（跨区域时）应该被限制。例如对象 |a| 在区域 A 里指向区域 B 里的对象 |b|，两个对象都没有被标记。考虑清除区域 B 之后将执行权返回 js 代码。存在可能 js 代码使得 |a| 变为 alive，这个时候需要标记 |b|，但是 |b| 已经被清除了。
- 所以如果存在这样的指针，那么 B 一定要在 A 之后清除。所以任何两个相互指向的区域必须同时被清除。使用 Tarjan 算法寻找一个图的强连通片进行清除。

- 对于没有析构的对象和**有允许后台执行的带析构函数的对象，在后台线程里清除**。

重置

在增量 gc 中下面情况尽管是小概率的，但是是可能的，会导致增量 GC 不再安全。在这样的情况下，收集器被`ResetIncrementalGC()`进行重置。如果程序处于标记状态，只需要停止标记。如果已经进入清除状态，继续执行清除工作知道当前需要被清除的区域被清除。在 reset 之后，一次非增量的 GC 开始执行。

Compacting GC

压缩 GC 在一次 GC 之后触发，作为最后一个时间片的一部分

- 选择相关的 arenas 进行压缩。
- 这些 arenas 的内容被移动到新的 arenas。
- 所有被移动对象的引用都要进行修改。

Collecting Atoms

- Atoms are collected differently from other GC things. They are contained in a special zone and things in other zones may have pointers to them that are not recorded in the cross compartment pointer map. Each zone holds a **bitmap** with the atoms it might be keeping alive, and atoms are only collected if
- 原子被回收的策略与其他 GC 对象不同。他们被包含在一个特殊的区域，这些区域里的对象可能存在指向他们带没有被记录在跨区域指针位图里的指针。每个区域维护一个位图，记录可能 alive 的原子，原子只有在他们不再被任意位图包含的时候才会被回收。见AtomMarking.cpp。

ChunkPool

定义 GC.cpp:723 —- 771

- 栈
- 无头节点的双向链表GC.cpp:740

```
1
2     //TODO
3     Iter;           //不知道数据类型 , GC.cpp:799
4
5     Chunk *pop();
6     void push(Chunk* chunk);
7     Chunk *remove(Chunk* chunk)
8
9     #ifdef DEBUG
10    bool verify();              //验证链表的正确性，指针有没有指对，计数是不是对的
11    #endif
```

8.3 数据结构

8.3.1 数据结构代码

Compartment 定义位置 js/src/wasm/wasmCompartment.h,cpp

lives in JSCompartment and contains the wasm-related per-compartment state.

tracks every live instance in the conpartment and must be notified, via regeisterInstance(), of any new WasmInstanceObject.

- instances_:InstanceVector — Instance 的数组 Q. Instance
- registerInstance(), unregisterInstance — 在一个实例被认为完全建立和有效之前必须 register，在被摧毁之前必须被 unregister
- instances() — 成员变量 get 函数 — 是一个 weak list，访问 instances()[i]->object() 出发一个读屏障 (read barrier)
- ensureProfilingLables() — 保证所有这个 JSCompartment 里的所有实例有一个 label
- addSizeOfExcludingThis() — 内存报告 Q. 这个不知道是啥
- sturct InstanceComparator — 实例比较器，其中的 operator() 方法用于比较两段代码的开始地址

Cell 类型 struct

GC cell 是所有 GC 事物的基类

一些方法概述

- isXXX() — 判断属性，有 tenured,marked(any, black, gray)
- is() — 判断类型
- as() — 转换类型
- address() — 地址
- chunk() — 获取 chunk
- 在 DEBUG 条件下 — dump() 和 isAligned()

TenuredCell

类型 class

8.4 StatisticsAPI

8.4.1 overview

环境变量 MOZ_GCTIMER 控制关于 GC 状态的文本信息输出。可能的值有 **none, stderr, stdout** 或一个文件名。如果是文件名的话，输出信息就写在文件里。

注 8.10 多线程 *+log*: *security.sandbox.content.level = 0*

browser preference javascript.options.mem.log — 可读性强的 GC 状态
browser preference javascript.options.mem.notify — JSON 编码的 GC 状态输出控制

8.4.2 JSON

顶级 JSON 对象包含的域:

- timestamp: 整数（毫秒）— GC 结束的时刻
- total_time: number(毫秒) — GC 使用的时间，是所有 slice 时间的综合
- compartments_collected: 整数 — 被收集的组件个数
- compartment_count: 整数 — GC 结束时系统里组件的个数

- max_parse:number(毫秒) — GC 期间的最长暂停时间
- nonincremental_reason:string — 增量 GC 结束或强制结束一个 slice 的原因
- allocated: 整数 (MB) — GC 开始时 js 堆的大小
- slices: 对象的数组 — 对象描述的是 slice
- times — 映射：从 phase names 到这个时间片花费的时间

最后两个域是需要整合工作的，包含很多子结构。比如 GC 的工作可以分为时间片 (phases)。另外，增量 GC 被分为一系列的 slice。

- slice:integer — slice 下标
- pause:number(毫秒) — slice 持续时间
- when:number(毫秒) — 开始时间：与第一个 slice 开始时间的相对时间
- reason:string — 描述开启这一 GC slice 的 API
- page_faults:integer — slice 期间发生的页错误
- start_timestamp:integer(毫秒) — 开始时间
- end_timestamp:integer(毫秒) — 结束时间
- times — 映射：phase names 到持续时间 n

8.5 public/GCAPI.h

gc 的运行模式

定义枚举 JSGCMode ,其中 JSGC_MODE_GLOBAL 执行全局 GC, JSGC_MODE_INCREMENTAL 执行增量 GC， JSGC_MODE_ZONE 在积累了大量的无用对象后执行 per-zone GC

Q. zone gc?

A. 可以对任意指定的 zone 集合进行 gc — 可以最小化收集时间 — 只收集最近的程序涉及的地方，忽略那些不可能被修改的堆空间 — PrepareZoneForGC 将 zone 标记为**将被收集**，PrepareForFullGC **选择所有 zone**， PrepareFprIncrementalGC 类似，相关的还有 SkipZoneForGC 、IsGCScheduled

gc 的触发模式

定义枚举 JSGCInvocationKind ，其中 GC_NORMAL 定义为正常的触发

启动 gc 的接口

- GCForReason(JSContext* cx, JSGCInvocationKind gckind, gcreason::Reason reason)
- StartIncrementalGC(JSContext* cx, JSGCInvocationKind gckind, gcreason::Reason reason, int64_t millis
- IncrementalGCSlice(JSContext* cx, gcreason::Reason reason, int64_t millis = 0)
- FinishIncrementalGC(JSContext* cx, gcreason::Reason reason)
- AbortIncrementalGC(JSContext* cx)

其他

debug 相关类 — GarbageCollectionEvent — 原因、各个 slice 的时间

Q. GarbageCollectionEvent(const GarbageCollectionEvent& rhs) = delete

progress — cycle_begin/end, slice_begin/end

8.6 内置 help() 函数的输出（部分）

[王瑞凯]

commands

`[$mozilla$/js/src]./build_debug/dist/bin/js` — 运行 js 引擎的 shell 程序

`[>] help()`

gc([obj] | 'zone' [, 'shrinking'])

执行一次 gc，如果指定了 obj，只回收这个 obj 的 zone。

如指定'zone'，回收所有使用 schedulegc 指定的 zone。

如果指定'shrinking'，执行一次 shrinking GC

gcparam(name [, value])

JS_[GS]etGCParameter 的包装器。可能的 name 有

- maxBytes
- maxMallocBytes
- gcBytes
- gcNumber
- mode
- unusedChunks
- totalChunks
- sliceTimeBudget
- markStackLimit
- highFrequencyTimeLimit
- highFrequencyLowLimit
- highFrequencyHighLimit
- highFrequencyHeapGrowthMax
- highFrequencyHeapGrowthMin
- lowFrequecyHeapGrowth
- dynamicHeapGrowth
- dynamicMarkSlice
- allocationThreshold
- minEmptyChunkCount
- maxEmptyChunkCount
- compactingEnabled
- refreshFrameSlicesEnabled

gczeal(level, [N])

- 0: (None) Normal amount of collection (resets all modes)
- 1: (Poke) Collect when roots are added or removed
- 2: (Alloc) Collect when every N allocations (default: 100)

- 3: (FrameGC) Collect when the window paints (browser only)
- 4: (VerifierPre) Verify pre write barriers between instructions
- 5: (FrameVerifierPre) Verify pre write barriers between paints
- 6: (StackRooting) Verify stack rooting
- 7: (GenerationalGC) Collect the nursery every N nursery allocations
- 8: (IncrementalRootsThenFinish) Incremental GC in two slices: 1) mark roots 2) finish collection
- 9: (IncrementalMarkAllThenFinish) Incremental GC in two slices: 1) mark all 2) new marking and finish
- 10: (IncrementalMultipleSlices) Incremental GC in multiple slices
- 11: (IncrementalMarkingValidator) Verify incremental marking
- 12: (ElementsBarrier) Always use the individual element post-write barrier, regardless of elements size
- 13: (CheckHashTablesOnMinorGC) Check internal hashtables on minor GC
- 14: (Compact) Perform a shrinking collection every N allocations
- 15: (CheckHeapAfterGC) Walk the heap to check its integrity after every GC
- 16: (CheckNursery) Check nursery integrity on minor GC

schedulegc([num | obj | string])

指定 num，num 个分配之后做 gc
指定 obj，回收 obj 对应的 zone
指定 string，回收 string 的 zone

selectforgc(obj1, obj2, ...)

标记这些对象

abortgc()

Abort the current incremental GC.

gcslice([n])

执行或继续一次增量 gc 的 slice，处理大约 n 个对象

gcstate()

Report the global GC state.
todo: 写一些脚本对其中的一些函数进行测试

8.7 splay.js 程序的阅读与测试

8.7.1 测试

命令

```
perf record ../build_debug/dist/bin/js wrk_run.js
perf report
```

结果

```
12.32%  js         js                          [.] js::GCMarker::drainMarkStack
10.95%  js         js                          [.] js::gc::ArenaCellIterImpl::getCell
3.92%  js         js                          [.] js::gc::Arena::finalize<JSObject>
1.89%  js         js                          [.] js::GCMarker::eagerlyMarkChildren
```

占用时间最长的前九个函数都与 GC 相关，说明 GC 应该是这个程序的性能瓶颈，对占用时间最长的几个函数代码进行阅读和分析。

GCMarker::drainMarkStack

```

if (budget.isOverBudget())
        return false;

    for (;;) {
        while (!stack.isEmpty()) {
            processMarkStackTop(budget);
            if (budget.isOverBudget()) {
                saveValueRanges();
                return false;
            }
        }

        if (!hasDelayedChildren())
            break;

        /*
         * Mark children of things that caused too deep recursion during the
         * above tracing. Don't do this until we're done with everything
         * else.
         */
        if (!markDelayedChildren(budget)) {
            saveValueRanges();
            return false;
        }
    }

    return true;
```

参数 — SliceBudget &budget

class SliceBudget(js/public/SliceBudget.h)：记录在给定的一段收集时间片 (collection slice) 中做了多少工作，因而我们可以在暂停太长时间之前返回。一些时间片 (slices) 不限制运行时间，其他是限制的。为了减少gettimeofday的调用，每 1000 个操作才会检查一次。

代码功能：循环对栈顶对象进行标记，直到栈空或因为有过深的引用时结束标记。如果结束时没有过深的引用，以为这所有应该被标记的对象都已经被标记了，返回 true；否则，应该被标记的对象中只有部分被标记，保存现场 (saveValueRanges)，返回 false。

处理标记栈的函数，只在两种情况下返回

1. overBudget — 堆里的对象太多了
2. hasDelayedChildren — 有一些 children 导致太深的递归

这里的 drainMarkStack 是断断续续地执行的（增量 gc 的特征），执行一段时间会后返回，让 js 代码执行。这里为了维持程序运行的正确性，在每次返回之前都会有一个现场保留的函数 — saveValueRanges

实现现场保留，在增量 gc 的下一个 slice 可以恢复当前运行状态

ArenaCellIterImpl::getCell

```

    TenuredCell* getCell() const {
        MOZ_ASSERT(!done());
        TenuredCell* cell = reinterpret_cast<TenuredCell*>(uintptr_t(arenaAddr) + thing);

        // This can result in a a new reference being created to an object that
        // an ongoing incremental GC may find to be unreachable, so we may need
        // a barrier here.
        if (needsBarrier)
            ExposeGCThingToActiveJS(JS::GCCellPtr(cell, traceKind));

        return cell;
    }
```

整个过程涉及到一个指针的转换和可能存在的一个 barrier 的生成。

Arena::finalize

```
    uint_fast16_t firstThing = firstThingOffset(thingKind);
    uint_fast16_t firstThingOrSuccessorOfLastMarkedThing = firstThing;
    uint_fast16_t lastThing = ArenaSize - thingSize;

    FreeSpan newListHead;
    FreeSpan* newListTail = &newListHead;
    size_t nmarked = 0;

    if (MOZ_UNLIKELY(MemProfiler::enabled())) {
        for (ArenaCellIterUnderFinalize i(this); !i.done(); i.next()) {
            T* t = i.get<T>();
            if (t->asTenured().isMarked())
                MemProfiler::MarkTenured(reinterpret_cast<void*>(t));
        }
    }

    for (ArenaCellIterUnderFinalize i(this); !i.done(); i.next()) {
        T* t = i.get<T>();
        if (t->asTenured().isMarked()) {
            ...
        } else {
            t->finalize(fop);
            JS_POISON(t, JS_SWEPT_TENURED_PATTERN, thingSize);
            TraceTenuredFinalize(t);
        }
    }

    if (nmarked == 0) {
        // Do nothing. The caller will update the arena appropriately.
```

```
30          MOZ_ASSERT(newListTail == &newListHead);
31          JS_EXTRA_POISON(data, JS_SWEPT_TENURED_PATTERN, sizeof(data));
32          return nmarked;
33      }
34
35      MOZ_ASSERT(firstThingOrSuccessorOfLastMarkedThing != firstThing);
36      uint_fast16_t lastMarkedThing = firstThingOrSuccessorOfLastMarkedThing - thingSize;
37      if (lastThing == lastMarkedThing) {
38          // If the last thing was marked, we will have already set the bounds of
39          // the final span, and we just need to terminate the list.
40          newListTail->initAsEmpty();
41      } else {
42          // Otherwise, end the list with a span that covers the final stretch of free things.
43          newListTail->initFinal(firstThingOrSuccessorOfLastMarkedThing, lastThing, this);
44      }
45
46      firstFreeSpan = newListHead;
47
48      return nmarked;
```

- 位置：js/src/jsgc.cpp line 414
- 作用：遍历整个 Arena，回收一部分对象，调整 FreeSpan
- 潜在优化：FreeSpan 管理模式

8.7.2 优化

Part IV

龙芯机器篇

9 cache 调研

9.1 龙芯 3A3000Cache 调研

[冉礼豪]

9.1.1 Cache 存储层次总览

- Cache 层次图

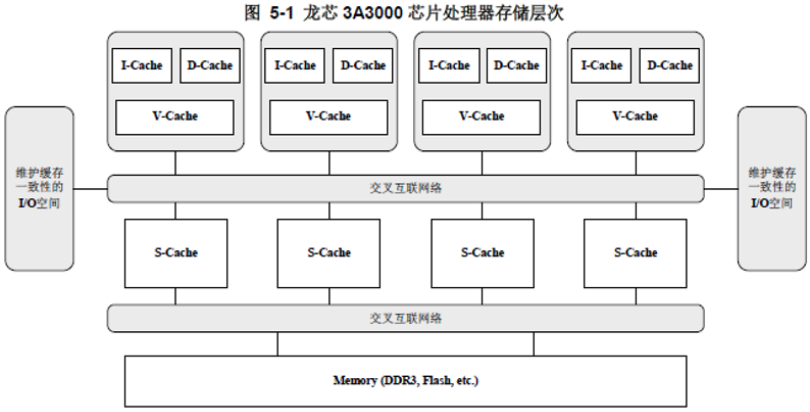


图 25: Cache 存储层次总览

- 结构总述
  根据各级缓存与处理器运算流水线的距离，由近及远，依次为：第一级的指令缓存 (Instruction-Cache，I-Cache) 和数据缓存 (Data-Cache，D-Cache)，第二级的牺牲缓存 (Victim-Cache，V-Cache)，第三级的共享缓存 (Shared-Cache，S-Cache)。其中 I-Cache、D-Cache 和 V-Cache 为每个处理器核私有，S-Cache 为多核及 I/O 共享。处理器核通过芯片内部及芯片之间的互联网络访问 S-Cache。
  具体细节参考龙芯 3A3000 用户手册龙芯 3A3000 用户手册第 59 页
- 各级 Cache 重要参数表

表 5-1 缓存参数

	指令缓存	数据缓存	牺牲缓存	共享缓存
容量	64KB	64KB	256KB	2MB/体 (芯片共 8MB)
相联度	4 路	4 路	16 路	16 路
行大小(line size)	512bit	512bit	512bit	512bit
索引(Index)	虚地址[13:6]	虚地址[13:6]	虚拟地址[13:6]	物理地址中的 11 比特 请参看 63 页 5.1.5 小节
标签(Tag)	物理地址[47:12]	物理地址[47:12]	物理地址[47:12]	物理地址[47:16]
替换策略	随机替换算法	LRU 替换算法	LRU 替换算法	LRU 替换算法
写策略		写回、写分配	写回、写分配	写回、写分配
校验方式	奇偶校验	SEC-DED ECC	SEC-DED ECC	SEC-DED ECC

图 26: 各级 Cache 重要参数表

9.1.2 各级 Cache 具体信息

- 一级指令 Cache
  - 缓存行数据结构

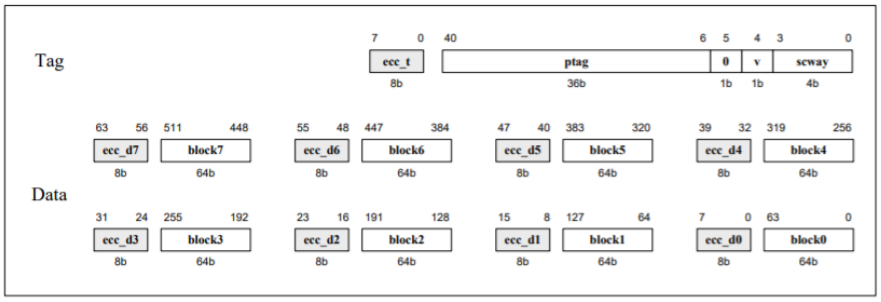


图 27: icache

  - 索引方式与实现
  - 数据校验方式与错误处理
    参考龙芯 3A3000 用户手册龙芯 3A3000 用户手册第 60 页

- 一级数据 Cache
  - 缓存行数据结构

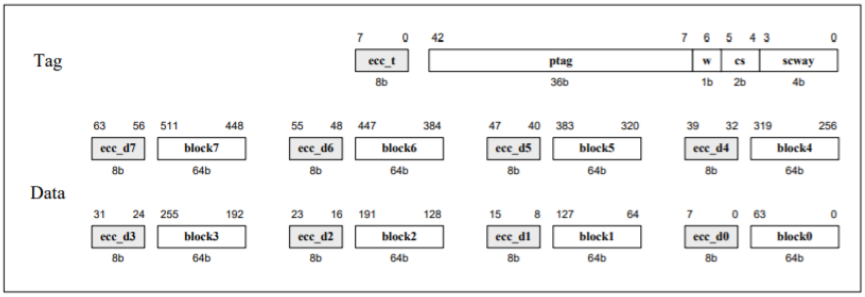


图 28: dcache

  - Tag 信息
  - 索引方式与实现
  - 数据校验方式与错误处理
    参考龙芯 3A3000 用户手册龙芯 3A3000 用户手册第 61 页

- 二级牺牲 Cache
  - 缓存行数据结构

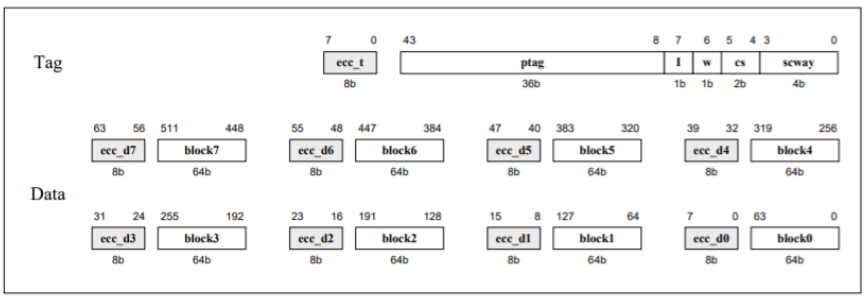


图 29: vcache

  - Tag 信息

  - 索引方式与实现
  - 数据校验方式与错误处理

    参考龙芯 3A3000 用户手册龙芯 3A3000 用户手册第 62 页

- 三级共享 Cache
  - 共享缓存的分体结构

表 5-2 三级共享缓存体选择位与索引地址

SCID_SEL 值	体选择位	索引地址
b0000	PAddr[7:6]	PAddr[18:8]
b0001	PAddr[9:8]	{PAddr[18:10], PAddr[7:6]}
b0010	PAddr[11:10]	{PAddr[18:12], PAddr[9:6]}
b0011	PAddr[13:12]	{PAddr[18:14], PAddr[11:6]}
b0100	PAddr[15:14]	{PAddr[18:16], PAddr[13:6]}
b0101~b1111	PAddr[18:17]	PAddr[16:6]

图 30: 分体结构

  - 共享缓存的锁机制
  - 共享缓存的缓存行结构

图 5-5 三级共享缓存行结构示意

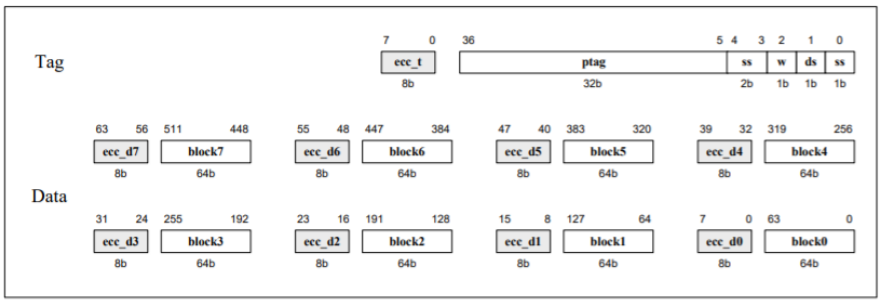


图 31: 三级共享缓存行结构示意

  - 共享缓存的校验

    参考龙芯 3A3000 用户手册龙芯 3A3000 用户手册第 63 页

9.1.3 缓存算法

- 非缓存算法
- 一致性缓存算法
- 非缓存加速算法

参考龙芯 3A3000 用户手册龙芯 3A3000 用户手册第 64-65 页

9.1.4 缓存一致性

- GS464E 实现了基于目录的缓存一致性协议，由硬件保证 I-Cache、D-Cache、V-Cache、S-Cache、内存以及来自 HT 的 IO 设备之间数据的一致性，无需软件利用 Cache 指令来维护缓存一致性。

  参考龙芯 3A3000 用户手册龙芯 3A3000 用户手册第 65 页

- 一级指令缓存与一级数据缓存间的一致性维护
  参考龙芯 3A3000 用户手册龙芯 3A3000 用户手册第 69 页
- 处理器与 DMA 设备间的缓存一致性维护
  参考龙芯 3A3000 用户手册龙芯 3A3000 用户手册第 69 页

9.1.5 缓存管理

- Cache 指令
  - 简述：参考龙芯 3A3000 用户手册龙芯 3A3000 用户手册第 66 页
  - 根模式下的 Cache 指令

op[4:0]	功能描述	目标 Cache
b00000	根据索引无效 Cache 行	I-Cache
b01000	根据索引写 Cache 行 Tag	I-Cache
b11100	根据索引写 Cache 行 Data	I-Cache
b00001	根据索引无效并写回 Cache 行	D-Cache
b00101	根据索引读 Cache 行 Tag	D-Cache
b01001	根据索引写 Cache 行 Tag	D-Cache
b10001	根据命中无效 Cache 行	D-Cache
b10101	根据命中无效并写回 Cache 行	D-Cache
b11001	根据索引读 Cache 行 Data	D-Cache
b11101	根据索引写 Cache 行 Data	D-Cache
b00010	根据索引无效并写回 Cache 行	V-Cache
b00011	根据索引无效并写回 Cache 行	S-Cache
b00111	根据索引读 Cache 行 Tag	S-Cache
b01111	根据地址取回并锁存 Cache 行	S-Cache

图 32: 根 cache 指令 1

op[4:0]	功能描述	目标 Cache
b01011	根据索引写 Cache 行 Tag	S-Cache
b10011	根据命中无效并写回 Cache 行	S-Cache
b11011	根据索引读 Cache 行 Data	S-Cache
b11111	根据索引写 Cache 行 Data	S-Cache

图 33: 根 cache 指令 2

  - 客模式下的 Cache 指令
    客模式下 CACHE 指令的使用受到根模式的控制，具体定义如下：
    * GuestCtl0.CG=0 时，使用任何 CACHE 指令都将触发客模式特权敏感指令例外 (GPSI)。
    * GuestCtl0.CG=1 时，使用 op[4:2]=1, 2, 6, 7 的 CACHE 指令将触发客模式特权敏感指令例外 (GPSI)。
    * GuestCtl0.CG=1，但 Diag.GCAC=0 时，使用 CACHE0、CACHE1、CACHE3 指令将触发客模式特权敏感指令例外 (GPSI)。

- 缓存初始化
  - 基于硬件的初始化

– 基于软件的初始化
参考龙芯 3A3000 用户手册龙芯 3A3000 用户手册第 67 页

9.1.6 Cache 的错误例外

当取指或 load/store 操作执行时发现 Cache 的 tag 或 data 出现校验错误时，触发该例外。该例外不可屏蔽。因为该例外涉及的错误在 Cache 中，所以采用专门的例外入口，位于非映射非缓存地址段。该例外仅在根模式下处理。

控制寄存器 Cause 的 ExcCode 域：

无

响应例外时的硬件状态更新过程：

```
CacheErr ← ErrorState
Status.ERL ← 1
if InstructionInBranchDelaySlot then
    ErrorEPC ← PC of the branch/jump
else
    ErrorEPC ← PC of the instruction
endif
if Status.BEV=1 then
    PC ← 0xFFFF.FFFF.BFC0.0200 + 0x100
else
    PC ← 0xFFFF.FFFF || EBase31..30 || 1 || EBase28..12 || 0x100
endif
```

图 34: 例外

参考龙芯 3A3000 用户手册龙芯 3A3000 用户手册第 77 页

10 性能事件 & 性能计数器

10.1 性能计数器

[**冉礼豪**]

10.1.1 处理器核性能计数器

每个 GS464E 的处理器核实现了 28 组性能计数器，均分在 FETCH 模块、RMAP 模块、ROQ 模块、FIX 模块、FLOAT 模块、MEMORY 模块和 CACHE2MEM 模块，每个模块各自包含 4 组性能计数器。每组性能计数器对应两个物理寄存器，一个用于计数（48 位计数器），一个用于控制计数。

GS464E 处理器核的性能计数器仍通过 MIPS 架构下的 CP0 Performance Counter 寄存器供软件配置和访问，但是访问的形式与传统 MIPS 处理器有些许差异。具体参见龙芯 3A3000 用户手册龙芯 3A3000 用户手册第 150 页

10.1.2 共享缓存性能计数器

每个 GS464E 的共享缓存体实现了 4 组 48 位的性能计数器。每组性能计数器对应两个物理寄存器，一个用于计数（48 位计数器），一个用于控制计数。每个性能计数器均可以统计该共享缓存体内发生的任何性能计数事件。

10.2 性能计数事件

10.2.1 简介

GS464E 定义的性能事件分为三大类：

- 用于分析程序在指令集层面体现出的特性。具体是统计流水线提交阶段不同类型指令各自的数量，从而得到程序动态执行指令类型的分布情况.
- 不命中次数、队列满次数、分支预测出错次数等基础事件外，GS464E 还新增了一批关于延迟周期数的统计，譬如分支误预测后清空前端流水线导致 regmap 流水级断流的周期数等.
- 用于为设计空间探索积累数据，出发点是为了优化微结构。譬如，当前双访存部件采用全功能双端口 RAM 实现，比单端口 RAM 相比开销大很多，因此加入了关于同一拍两条 load 操作 dcache RAM 冲突的次数的统计，在现有结构上这种冲突其实不会造成流水线塞，因此并不影响程序的性能

10.2.2 具体定义

处理器核性能事件:

表 8-2 处理器核性能计数器事件定义

事件号	事件描述
	FETCH 模块
1	Inst Queue 全空的周期数
2	每周期写入 Inst Queue 的指令数
3	前端流水线阻塞周期数（进入 Inst Queue 的指令数等于 0）
4	进入 Inst Queue 的指令数等于 1
5	进入 Inst Queue 的指令数等于 2
6	进入 Inst Queue 的指令数等于 3
7	进入 Inst Queue 的指令数等于 4
8	进入 Inst Queue 的指令数等于 5

151

图 35:

共享缓存性能事件:

11 loongson 处理器体系架构

（本文档中出现的图片'_' 前的前缀标示了图片的来源，其中第一个数字表示龙芯手册的编号（1 或 2，上册或下册），第二个数字表示手册中的章节号。若前缀为"ex"，即表示图片来自其他来源）

9	进入 Inst Queue 的指令数等于 6
10	进入 Inst Queue 的指令数等于 7
11	进入 Inst Queue 的指令数等于 8
12	cache 空间下因为跨越 cacheline 边界使得进入 InstQueue 的指令数小于 8
13	由于 Inst Queue 满造成的前端流水线阻塞周期数
14	每周期译码指令数
15	每周期来自 Loop Buffer 的译码指令数
16	从 Loop buffer 取指令的 Loop 个数
17	识别出的 Loop 个数（包含可以放入 Loop Buffer 和无法放入 Loop Buffer 的）
18	每周期译码的分支指令数等于 0
19	每周期译码的分支指令数等于 1
20	每周期译码的分支指令数等于 2
21	由于 Icache Miss 造成的前端流水线阻塞
22	由于 BrBTB 未能成功预测 taken branch 造成的前端流水线阻塞
24	Icahe 模块发起且被 missq 接收的 Icache miss 次数
26	ITLB miss 但在 TLB 命中的次数
27	ITLB 被 flush 的次数
	RMAP 模块
64	资源分配被阻塞
65	GR 重命名资源满阻塞
66	GR 重命名资源满假阻塞（待进入指令无需要定点重命名资源的）
67	FR 重命名资源满阻塞
68	FR 重命名资源满假阻塞（待进入指令无需要浮点重命名资源的）
69	FCR 重命名资源满阻塞
70	FCR 重命名资源满假阻塞（待进入指令无需要 FCR 重命名资源的）
71	ACC 重命名资源满阻塞
72	ACC 重命名资源满假阻塞（待进入指令无需要 ACC 重命名资源的）
73	DSPCtrl 重命名资源满阻塞
74	DSPCtrl 重命名资源满假阻塞（待进入指令无需要 DSPCtrl 重命名资源的）
75	BRQ 满阻塞
76	BRQ 满假阻塞（待进入指令无需要进入 BRQ 的）
77	FXQ 满阻塞
78	FXQ 满假阻塞（待进入指令无需要进入 FXQ 的）
79	FTQ 满阻塞
80	FTQ 满假阻塞（待进入指令无需要进入 FTQ 的）
81	MMQ 满阻塞
82	MMQ 满假阻塞（待进入指令无需要进入 MMQ 的）
83	CP0Q 满阻塞
84	CP0Q 满假阻塞（待进入指令无需要进入 CP0Q 的）
85	ROQ 满阻塞
86	完成资源分配阶段的 NOP 类指令数量

图 36:

11.1 龙芯处理器概述

[**刘时**]

龙芯处理器主要包括三个系列，三个系列并行地发展。其中，龙芯 3 号多核处理器系列主要面向服务器和高性能机应用。

龙芯 3 号多核系列处理器基于可伸缩的多核互连架构设计，可通过高速 I/O 接口实现多芯片的互连以组成更大规模的系统，其互联结构如图：

具体来说，其节点结构如下图所示：

每个结点有两级 AXI 交叉开关（X1 Switch，X2 Switch）连接处理器、Cache、内存控制器及 I/O 控制器等，同时，X1 Switch 上还有 4 对端口用作多片互连。

龙芯 3A3000/3B3000 的主要技术指标如下：

- 单节点 4 核架构，28nm 工艺，1.2 ~ 1.5GHz 主频
- 片内集成 4 个 64 位的四发射超标量 GS464e 高性能处理器核

87	每周期从 regmap 发射至各发射队列的操作数
88	例外(不含分支误预测)清空流水线的开销（regmap 流水级被例外清空后至首条指令到达 regmap 流水线）
89	分支预测错清空流水线的开销
	ROQ 模块
128	内部流水线时钟
129	每周期提交指令数
130	提交的 ALU 操作
131	提交的 FALU 操作
132	提交的 Memory/CP0/定浮点互换操作
133	提交的 load 操作
134	提交的 store 操作
135	提交的 LL 类操作
136	提交的 SC 类操作
137	提交的非对齐 load 操作
138	提交的非对齐 store 操作
139	所有例外和中断的次数
140	中断的次数
141	从 ROQ 接收到中断信号到产生中断例外
142	从 ROQ 接收到中断信号到中断例外处理程序第一条指令进入 ROQ
143	虚拟机例外次数
144	地址错例外次数
145	TLB 相关例外次数
146	TLB refill 例外次数
147	TLB refill 例外的处理时间（TLB refill 例外清空流水线开始到 TLB refill 例外的 ERET 返回）
148	brq 提交的分支指令
149	brq 提交的 jump register 分支指令
150	brq 提交的 jump and link 分支指令
151	brq 提交的 branch and link 分支指令
152	brq 提交的 bht 分支指令
153	brq 提交的 likely 分支指令
154	brq 提交的 not taken 的分支指令
155	brq 提交的 taken 的分支指令
156	brq 提交的预测错的分支指令
157	brq 提交的预测错的 jump register 分支指令
158	brq 提交的预测错的 jump and link 分支指令
159	brq 提交的预测错的 branch and link 分支指令
160	brq 提交的预测错的 bht 分支指令
161	brq 提交的预测错的 likely 分支指令
162	brq 提交的预测错的 not taken 的分支指令
163	brq 提交的预测错的 taken 的分支指令
	FIX 模块

图 37:

- 片内集成 8MB 的分体共享三级 Cache(由 4 个体模块组成，每个体模块容量为 2MB)
- 通过目录协议维护多核及 I/O DMA 访问的 Cache 一致性
- 片内集成 2 个 64 位带 ECC，667MHz 的 DDR2/3 控制器
- 3B3000 片内集成 2 个 16 位 1.6GHz 的 HyperTransport 控制器（以下简称 HT)；3A3000 片内 HT1 为 16 位 1.6GHz 的 HT 控制器，HT0 不可用
- 每个 16 位的 HT 端口拆分成两个 8 路的 HT 端口使用
- 片内集成 32 位 33MHz PCI
- 片内集成 1 个 LPC、2 个 UART、1 个 SPI、16 路 GPIO 接口

> 超标量：利用多套功能单元（算术逻辑单元等），实现在一个时钟周期内，通过不同的指令在多个功能原件中并行执行，执行多条指令的技术。“四发射”指支持四条指令同时使用一种功能单元执行计算。下图是一个双发射超标量流水线的示例。

> ECC（Error-correcting code）：检错校验功能
>

事件号	事件描述
342	load 命中次数
343	fwdbus2 次数
344	fwdbus5 次数
345	fwdbus 总次数，fwdbus2+fwdbus5
346	因 load、store 地址冲突导致的访存操作回滚次数（dwaitstore）
347	因 load、store 地址冲突导致的例外次数（mispec）
348	cp0qhead 因 dcachewrite 不顺利而回滚的次数
349	cp0q 发出 dmemread 请求次数
350	cp0q 发出 duncache 请求次数
351	resbus2 挤占 resbus5 次数，LQ、LQC1 等存在两个 dest 的访存操作
352	软件预取在 L1 Dcache 命中次数
353	store 软件预取在 L1 dcache 命中次数
354	store 软件预取在 L1 dcache miss 次数
355	load 软件预取在 L1 dcache 命中次数
356	load 软件预取在 L1 dcache miss 次数
357	store 软件预取在 L1 dcache 因 share state 不命中的次数
358	specfwdbus2 次数
359	specfwdbus5 次数
360	specfwdbus 次数：specfwdbus2+specfwdbus5
384	数据 load 请求访问 vcache 次数
385	数据 store 请求访问 vcache 次数
386	数据请求访问 vcache 次数
387	指令请求访问 vcache 次数
388	vcache 访问次数
389	软件预取访问 vcache 的次数
390	vcache load 命中次数
391	vcache store 命中次数
392	vcache 数据命中次数
393	vcache 指令命中次数
394	vcache 命中次数
395	vcache 软件配置预取命中次数
396	vcache load 失效次数
397	vcache store 失效次数
398	vcache 数据失效次数
399	vcache 指令失效次数
400	vcache 失效次数
401	vcache 软件配置预取失效次数
402	vcache 被 extreq 操作无效掉有效块的次数
403	vcache 被 wtbk 操作降级有效块的次数
404	vcache 被 INV 操作无效掉有效块的次数
405	vcache 被 INVWTBK 无效掉有效块的次数

图 38:

```
> HyperTransport（HT） 控制器：用于实现外部设备连接及多片互连的控制器。
>
> PCI、LPC、UART、SPI、GPIO：各种 I/O 相关接口。）
>
> JTAG（Joint Test Action
```

Group）是一套测试印刷电路板的标准，EJTAG 是其在 MIPS 指令集上的实现及拓展。TAP 即测试访问接口（Test Access Port）。

11.2 矩阵加速处理器

3A3000/3B3000 中内置两个独立于处理器核心的矩阵加速处理器，继承在芯片的两个 HT 控制器内部，可实现矩阵的转置和缓存预取操作。为了提高转置效率，其内部有一个 32 行的缓冲区，可以一行读入多行数据，这样就可以一次输出多个目标矩阵元素。

192	fxq 无发射
193	fxq 发射执行操作数
194	fxq 发射到 FU0 功能部件执行的操作数
195	fxq 发射到 FU1 功能部件执行的操作数
196	FU0 中定点乘法部件处于执行状态
197	FU0 中定点除法部件处于执行状态
198	FU1 中定点乘法部件处于执行状态
199	FU1 中定点除法部件处于执行状态
	FLOAT 模块
256	ftq 无发射
257	ftq 发射执行操作数
258	ftq 发射到 FU3 功能部件执行的操作数
259	ftq 发射到 FU4 功能部件执行的操作数
260	FU3 空闲，FU4 满，但 ftq 只有 FU4 待发射项
261	FU4 空闲，FU3 满，但 ftq 只有 FU3 待发射项
262	每周期发射标量浮点操作数
263	每周期发射的 64 位多媒体加速指令数(指令名含“GS”前缀)
264	每周期发射的 64 位多媒体加速指令数(指令名不含“GS”前缀)
272	FU3 中浮点除法/开方部件处于执行状态
274	FU4 中浮点除法/开方部件处于执行状态
	MEMORY 模块
320	mmq 无发射
321	mmq 发射执行操作数
322	mmq 中，每拍发射 FU2 指令
323	mmq 中，每拍发射 FU5 指令
324	load 发射次数
325	store 发射次数
326	源操作数至少有一个浮点的访存指令数量
327	同时具有定点和浮点操作数的指令的发射次数
329	wait_first 阻塞的周期数
330	SYNC 操作阻塞的周期数
331	stall_issue 阻塞的周期数
332	软件预取操作发射
333	新发射的访存操作堵塞 store 操作写入 dcache 的周期数
334	同一拍两个 load 发生 bank 冲突
337	dcachewrite0 和 1 同时有效的次数
338	执行成功的 SC 类指令次数
339	store 指令 dcahe miss 次数（包括不命中和非 EXC 状态）
340	store 指令因 dcache shared 状态导致的 dcache miss 次数
	CACHE2MEM 模块
341	store 指令 dcache 命中次数

图 39:

> 问题：我们实验平台的处理器型号？3A3000 只有一个 HT 控制器可用，是否说明只有一个矩阵加速处理器？该处理器是 0 号还是 1 号转置模块?

矩阵处理编程接口说明，矩阵处理寄存器地址、内容说明见手册上册 34 ~ 36 页。

11.3 处理器核间中断与通信

每个处理器核配备 8 个核间中断寄存器以支持在多核 BIOS 启动以及操作系统运行时进行处理器核间中断与通信。

各处理器地址与功能描述间手册上册 37 ~ 38 页。多处理器系统中寄存器地址与其节点基地址存在固定的偏移关系。

406	处理器核对外总线读请求次数
407	处理器核对外总线写请求次数
408	带有写数据的总线写请求次数
409	总线读请求因与总线写请求地址冲突而被阻塞
410	missq 处理的 WTBK 请求数量
411	missq 处理的 INVWTBK 请求数量
412	missq 处理的 INV 请求数量
413	missq 处理的 INV 类（上述 3 个）请求数量
414	refill 的总次数(包括 exreq 和 replace+refill)
415	refill 的针对 icache 的总次数
416	refill 的针对 dcache 的总次数
417	refill(replace+refill)的次数
418	refill 一个 dcache　shared 块的次数
419	refill 一个 dcache　exc 块的次数
420	refill 数据的总次数（replace+refill）
421	refill 指令的总次数（replace+refill）
422	dcache 替换出来一个有效块的次数
423	dcache 替换出来一个 shared 块的次数
424	dcache 替换出来一个 exc 块的次数
425	dcache 替换出来一个 dirty 块的次数
426	icache 替换出有效数据的次数
427	vcache 替换次数
428	vcache 替换出一个有用块的次数
429	vcache 替换出一个 shared 块的次数
430	vcache 替换出一个 exc 块的次数
431	vcache 替换出来一个 dirty 块的次数
432	vcache 替换出有用 dc 块的次数
433	vcache 替换出有用 ic 块的次数
434	累计每一拍未从 scache 返回的 load 请求数量（missq 最多只有 15 项用于处理 scache 请求）
435	累计每一拍未从 scache 返回的 store 请求数量
436	累计每一拍未从 scache 返回的取指请求数量
437	送出的 sc read 的总个数
438	送出的 scread 中 load 的总个数
439	送出的 scread 中 store 的总个数
440	scread 数据访问总个数
441	scread 指令访问总个数
442	scread 非预取总个数
443	scread 非预取数据 load 总个数
444	scread 非预取数据 store 总个数
445	scread 非预取数据访问总个数
446	scread 非预取取指访问总个数

图 40:

11.4 I/O 中断

3A3000/3B3000 芯片最多支持 32 个中断源，可由中断路由选择目标处理器核心中断脚。

中断控制寄存器地址、描述，中断路由寄存器地址与描述见手册上册 41 ~ 42 页。

11.5 GS464e 处理器核概述

GS464E 是一款实现了 LoongsonISA 指令集的通用 RISC 处理器核。GS464E 的指令流水线每个时钟周期取四条指令译码，并且动态地发射到六个全流水的功能部件中。指令在保证依赖关系的前提下可以乱序发射执行，所有指令按照程序中的顺序进行提交以保证精确例外和访存顺序执行。

GS464E 中为解决指令相关和数据相关问题，引入了多种乱序执行技术，包括寄存器重命名技

事件号	事件描述
447	送出的 scread 中预取的总个数
448	送出的 scread 中 load 预取的个数
449	送出的 scread 中 store 预取的个数
450	scread 预取数据访问总个数
451	scread 预取指令访问总个数
452	missq 处理的软件预取请求个数
453	missq 发出的 scwrite 个数
454	missq 因 replace 操作发起的 scwrite 个数
455	missq 因 invalid 操作发出的 RESP 类 scwrite 个数
456	missq 因 replace 操作发起的 scwrite 操作，且 replace 有效块
457	missq 真正接受请求个数 miss_en
458	missq 真正接受 load 请求个数 miss_en
459	missq 真正接受 store 请求个数 miss_en
460	missq 真正接受的数据访问的个数
461	missq 真正接受的指令访问的个数
462	missq 占项情况(missq 非空项)
463	missq 普通访问占项情况
464	missq 取指访问占项情况
465	missq 外部请求占项情况
466	missq 预取请求占项情况
467	missq 占项拍数（missq 存在有效项的拍数，即 missq 不空时间）
468	missq 普通访问占项拍数
469	missq 中取指访问占项拍数
470	missq 外部请求占项拍数
471	missq 预取请求占项拍数
472	missq 满计数（missq 不能接受普通访问，missq 有效项不小于 15）
473	load 请求在 missq 中碰到预取的次数
474	load 请求在 missq 中碰到预取 pre_scref 的次数
475	load 请求在 missq 中碰到预取 pre_wait 的次数
476	load 请求在 missq 中碰到预取 pre_rdy 的次数
477	store 请求在 missq 中碰到预取 pre_scref 且 load 操作的次数
478	store 请求在 missq 中碰到预取 pre_rdy 且 state=shard 次数
479	store 请求在 missq 中碰到预取 pre_wait 且 load 操作次数
480	store 请求在 missq 中碰到预取 pre_scref 且 store 操作的次数
481	store 请求在 missq 中碰到预取 pre_rdy 且 state=exc 次数
482	store 请求在 missq 中碰到预取 pre_wait 且 store 操作的次数
483	store 请求在 missq 中碰到预取的次数（包括命中在 store 预取和命中在 load 预取）
484	store 请求在 missq 中碰到有效预取的次数（命中在 store 预取）
485	所有请求在 missq 中碰到预取的次数（load+store）
486	所有请求在 missq 中碰到 pre_scref 预取的次数（laod+store）
487	所有请求在 missq 中碰到 pre_rdy 预取的次数（load+store）

图 41:

术、动态调度技术和转移预测技术。

寄存器重命名，指通过将一组指令中使用的寄存器更改以实现更好的乱序执行效果的技术。举例:

```
R1=M[1024]
R1=R1+2
M[1032]=R1
R1=M[2048]
R1=R1+4
M[2056]=R1
```

```
R1=M[1024]
```

事件号	事件描述
488	所有请求在 missq 中碰到 pre_wait 预取的次数（load+store）
489	取指请求在 missq 中碰到预取的次数
490	取指请求在 missq 中碰到 pre_scref 预取的次数
491	取指请求在 missq 中碰到 pre_rdy 预取的次数
492	取指请求在 missq 中碰到 pre_wait 预取的次数
495	数据和取指在 missq 中碰到 pre_rdy 预取的次数
496	数据和取指在 missq 中碰到 pre_wait 预取的次数
497	硬件 load 预取请求被 Scache cancel 的次数 [15]
498	硬件 store 预取请求被 Scache cancel 的次数
499	硬件数据访问预取请求被 Scache cancel 的次数
500	硬件取指预取请求被 Scache cancel 的次数
501	硬件预取请求被 Scache cancel 的次数
502	硬件 load 预取的个数
503	硬件 store 预取的个数
504	硬件数据访问预取的个数
505	硬件取指预取的个数
506	硬件预取的个数
507	tagged 触发的 load 预取个数
508	miss 触发的 load 预取个数
509	tagged 触发的 store 预取个数
510	miss 触发的 store 预取个数
511	tagged 触发的数据访问预取个数
512	miss 触发的数据预取个数
513	tagged 触发的指令预取个数
514	miss 触发的指令预取个数
515	tagged 触发的预取个数
516	miss 触发的预取个数
517	被 missq 接受的 load 预取个数
518	被 missq 接受的 store 预取个数
519	被 missq 接受的数据访问预取个数
520	被 missq 接受的指令预取个数
521	被 missq 接受的预取个数（重复请求不进入 missq）
522	从 scache 回来的有效 load 预取个数
523	从 scache 回来的有效 store 预取个数
524	从 scache 回来的有效数据访问预取个数
525	从 scache 回来的指令预取个数
526	从 scache 回来的有效预取个数（pre_scref->rdy || pre_scref->pre_rdy）
527	能进入 pre_rdy 的 load 预取个数
528	能进入 pre_rdy 的 store 预取个数
529	能进入 pre_rdy 的数据访问预取个数

图 42:

```
R1=R1+2
M[1032]=R1
R2=M[2048]
R2=R2+4
M[2056]=R2
```

寄存器重命名可解决 WAR 和 WAW 相关，并用于例外和错误转移预测引起的精确现场恢复。

动态调度根据指令操作数准备好的次序而不是指令在程序中出现的次序来执行指令，减少了 RAW 相关引起的阻塞。GS464E 使用指令队列来实现动态调度。定点、浮点和访存指令有各自的指令队列，指令提交处还有一个输出队列。前者使只有数据准备好的指令被执行，后者使指令按输入顺序提交。

转移预测通过预测转移指令是否成功跳转来减少由于控制相关引起的阻塞。

事件号	事件描述
530	能进入 pre_rdy 的指令预取个数
531	能进入 pre_rdy 的预取个数（pre_scref -> pre_rdy）
532	累计每一拍处于 pre_rdy 的 load 预取请求个数
533	累计每一拍处于 pre_rdy 的 store 预取请求个数
534	累计每一拍处于 pre_rdy 的数据访问预取请求个数
535	累计每一拍处于 pre_rdy 的指令预取请求个数
536	累计每一拍处于 pre_rdy 的请求个数
537	累计每一拍处于 pre_scref 且被正常 load 请求命中的预取个数
539	累计每一拍处于 pre_scref 且被正常 store 请求命中的预取个数
540	累计每一拍处于 pre_scref 且被正常数据访问请求命中的预取个数
541	累计每一拍处于 pre_scref 且被正常取指请求命中的预取个数
542	累计每一拍处于 pre_scref 且被正常访问命中的预取个数
543	在 pre_scref 状态就被 load 访问 hit 的预取个数
544	在 pre_scref 状态就被 store 访问 hit 的预取个数
545	在 pre_scref 状态就被数据访问 hit 的预取个数
546	在 pre_scref 状态就被取指访问 hit 的预取个数
547	在 pre_scref 状态就被 hit 的预取个数，即从 pre_scref -> rdy 的次数
548	从 pre_scref 状态回到 miss 状态的 load 个数
553	在 pre_wait 状态就被 load 访问 hit 的预取个数
554	在 pre_wait 状态就被 store 访问 hit 的预取个数
555	在 pre_wait 状态就被数据访问 hit 的预取个数
556	在 pre_wait 状态就被取指访问 hit 的预取个数
557	在 pre_wait 状态就被 hit 的预取个数，即从 pre_wait -> pcmiss 的次数
558	因 missq 不能接受正常访问而被替换出去的处于 pre_wait 状态的预取项
559	因 missq 不能接受正常访问而被替换出去的处于 pre_rdy 状态的预取项
560	预取项被 INV 的次数
561	累计每一拍 load 预取占项情况
562	累计这一拍 load 预取是否占项
563	累计每一拍 store 预取占项情况
564	累计这一拍 store 预取是否占项
565	累计每一拍数据预取占项情况
566	累计这一拍数据预取是否占项
567	累计每一拍取指预取占项情况
568	累计这一拍取指预取是否占项
569	累计 load 预取在 pre_scref 和 pre_rdy 命中个数
570	累计 store 预取在 pre_scref 和 pre_rdy 命中个数
571	累计数据预取在 pre_scref 和 pre_rdy 命中个数
572	累计取指预取在 pre_scref 和 pre_rdy 命中个数
573	累计预取在 pre_scref 和 pre_rdy 命中个数

图 43:

具体到 GS464E 实现以上乱序执行技术引入的结构，可以参考手册下册第一页。

GS464E 的基本流水线包括 PC、取指、预译码、译码 I、译码 II、寄存器重命名、调度、发射、读寄存器、执行、提交 I、提交 II 等 12 级。其中每一级操作如下：

- PC：生成下一拍取指所需程序计数器 PC 值
- 取指：用 PC 值访问指令 Cache 和指令 TLB。如果两者都命中，取 8 条指令到指令寄存器 IR
- 预译码：对转移指令译码，预测跳转方向
- 译码 I：将预译码结果存入指令队列
- 译码 II：将 IR 中的四条指令转换为处理器的内部指令送往寄存器重命名模块
- 寄存器重命名：为逻辑目标寄存器分配新的物理寄存器，并将逻辑源寄存器映射到最近分配给它的物理寄存器
- 调度：将重命名后的指令分配到定点或浮点保留队列中等待执行，同时送到 ROQ 用于顺序提

8.2.2 共享缓存性能计数事件定义

表 8-3 共享缓存性能计数器事件定义

事件号	事件描述
0	因 ctrl 中没有开关标记，因此配置为 0 表示没有开。
1	收到的所有 request 请求的个数
2	收到的 cachable 的 request 请求的个数
3	收到的所有非 DMA 操作的 cachable 的 request 请求
4	收到的所有非预取属性的 cachable 的 request 请求
5	收到的所有 cached 的 REQ_READ 请求
6	收到的所有 cached 的 REQ_WRITE 请求
7	收到的所有的非预取属性的 cached 的 REQ_READ
8	收到的所有的非预取属性的 cached 的 REQ_WRITE
9	收到的所有 cached 的 DMAREAD 请求
10	收到的所有 cached 的 DMAWRITE 请求
11	收到的所有 uncached 的 DMAREAD
12	收到的所有 uncached 的 DMAWRITE
13	收到的所有 DMA 请求个数
14	收到的所有类型是 scache_prefx 的请求个数
15	Scache 硬件自己生成的预取被接受的个数
16	收到的所有类型是 store_fill_full 的请求个数
17	收到的所有针对一致性请求的响应
18	收到的所有针对一致性响应并且数据为 dirty 的个数
19	收到的上一级 cache 主动替换的写回
20	收到的上一级主动替换且数据为 dirty 的个数
21	由 cached 访问导致的对 memory 的读请求个数
22	由 cached 访问导致的对 memory 的写请求个数
23	查询 scache 结果为 miss 的次数
24	查询 scache 结果为 miss 的所有 REQREAD 个数
25	查询 scache 结果为 miss 的所有 REQWRITE 个数
26	查询 scache，scache 命中但 pagecolor 不命中的个数
27	REQREAD 查询 scache，结果命中在一个 clean 块
28	REQREAD 查询 scache，结果命中在别人的 EXC 块
29	REQWRITE 查询 scache，结果命中在 clean 块
30	REQWRITE 查询 scache，结果命中在别人的 EXC 块
31	WRITE 查询 scache，结果命中在一个正在被多个核 shared 的块中
32	Cached DMAREAD 查询 scache，结果命中
33	Cached DMAWRITE 查询 scache，结果不命中
34	Cached DMAWRITE 查询 scache，结果命中，并需要对 CPU 发出 INVALIDATION
35	向 CPU 发出的一致性请求中 INV 请求的个数
36	向 CPU 发出的一致性请求中 WTBK 请求的个数
37	向 CPU 发出的一致性请求中 INVWTBK 请求的个数

图 44:

交；转移指令和访存指令同时还被送到转移队列和访存队列

- 发射：为每个功能部件从保留队列中选出操作数都准备好的指令；侦听总线等到其他指令准备好
- 读寄存器：从物理寄存器堆读指令的原操作数，送至对应的功能部件
- 执行：执行指令并把结果写回寄存器堆；同时通知保留队列和寄存器重命名表释放资源。对于定点乘除法、浮点以及访存等复杂指令，这一阶段需要多拍。
- 提交 I：根据 ROQ，选择可以提交的指令；每拍最多可以提交 4 条指令
- 提交 II：将提交的指令送至寄存器重命名表确认目的寄存器的重命名关系并释放被分配的物理寄存器；将存数指令送往访存队列，使之写入 Cache 或内存。

主要特点：

- MIPS64 兼容的 LoongISA 指令集
- 12 级流水深度
- 取指宽度高达 8 条指令/周期
- 发射宽度高达 (2 定点 +2 浮点 +2 访存)/周期

- 四发射超标量结构，两个定点、两个浮点、两个访存部件
- 每个浮点部件都支持全流水 64 位/双 32 位浮点乘加运算
-
- 访存部件支持 128 位存储访问，虚地址为 64 位，物理地址为 48 位
- 支持寄存器重命名、动态调度、转移预测等乱序执行技术
- 64 项全相联外加 8 路组相连 1024 项，共计 1088 项 TLB，64 项指令 TLB
- TLB 支持 4^i KB 可变页大小（i=1，2，……，10）
- 一级指令 Cache 和数据 Cache 大小各为 64KB，4 路组相联
- Victim Cache 作为私有二级 Cache，大小为 256KB，16 路组相连
- 三级多核共享 SCache，2MB，16 路组相连
- 支持 Non-blocking 访问及 Load-Speculation 等访存优化技术
- 支持 Cache 一致性协议，可用于片内多核处理器
- 指令 Cache 实现奇偶校验，数据 Cache 实现 ECC 校验
- 支持标准的 EJTAG 调试标准，方便软硬件调试
- 标准的 128 位 AXI 接口

```
> 将第一组指令修改为第二组后，执行效果不变，但使得后三条指令不依赖R1寄存器，可以与前三条并行执行，提高并行度。
>
> 直接映射TLB：每个页只能映射到一个对应TLB页，电路简单，效果较差。
>
> 全相联TLB：每个页可以映射到任一 TLB 页，效果最好，电路最复杂；
>
> 组相联TLB：页表和TLB都分组，每组页表项映射到一组TLB页，组内全相联，组间直接映射。
>
> AXI：第三代 ARM AMBA 总线标准，用于运行单元和 SoC 互连和管理。
```

物理结构：

逻辑结构：