Resin 源码解读

作者: Edisonpeng

2007 年 8 月 10 日

整理的还不是很完整,看了不要骂,后面我会继续改进:-)!~!

Preface

从日志分析开始

启动日志

首先要开启 debug 日志,只需要在 RESIN_HOME/conf/resin.conf 中加入如下配置节:

```
<log name="" level='finer' path="log/debug.log" timestamp="[%H:%M:%S.%s]" rollover-period="1D"/>
```

通过 debug 日志,我们可以查看 resin 运行时的处理流程等.下面为 resin 从启动到整个加载过程完毕的 debug.log 输出:

```
[16:50:23.036]ServerController$5924809[] initializing
[16:50:23.041]ServletServerMBean[resin:type=Server,name=default] registered in
MBeanContext[sun.misc.Launcher$AppClassLoader@198dfaf]
[16:50:23.041]ServerController$5924809[] initialized
[16:50:23.041]Resin[] initialized

[16:50:23.042]Resin[] active

[16:50:23.157]MBeanServerDelegateMBean[JMImplementation:type=MBeanServerDelegate] registered in
MBeanContext[EnvironmentClassLoader$20863188[servlet-server:]]
[16:50:23.157]ServletServerMBean[resin:name=default,type=Server] registered in
MBeanContext[EnvironmentClassLoader$20863188[servlet-server:]]
[16:50:23.158]ThreadPoolMBean[resin:type=ThreadPool] registered in
MBeanContext[EnvironmentClassLoader$20863188[servlet-server:]]
[16:50:23.160]ServerController$5924809[] starting

[16:50:23.160]Server[] initializing

[16:50:23.162]Server[] root-directory=/usr/local/resin_test
[16:50:23.162]ServletServerMBean[resin:type=CurrentServer] registered in
MBeanContext[EnvironmentClassLoader$20863188[servlet-server:]]
[16:50:23.308]ClusterClientMBean[resin:type=ClusterClient,Server=default,Cluster=default,host=127.0.0.1,port=6803]
registered in MBeanContext[EnvironmentClassLoader$20863188[servlet-server:]]
[16:50:23.312]ClusterMBean[resin:type=Cluster,Server=default,name=default] registered in
MBeanContext[EnvironmentClassLoader$20863188[servlet-server:]]
[16:50:23.322]resin:import '/usr/local/resin_test/conf/app-default.xml'
[16:50:23.514]HostExpandDeployGenerator[/usr/local/resin_test/hosts] redeploy false

[16:50:23.532]Server[] starting

#系统相关信息
[16:50:23.532]
[16:50:23.533]Linux 2.4.31 i386
```

[16:50:23.533]Java 1.5.0_01-b08, 32, mixed mode, ANSI_X3.4-1968, en, Sun Microsystems Inc.
[16:50:23.534]resin.home = /usr/local/resin_test
[16:50:23.534]server.root = /usr/local/resin_test
[16:50:23.535]

#启动 Port, Port 是 resin 中的主处理线程,如下这个是所有 webapp 的入口 Port,端口为 8081
#Resin 内部实现了 JMX 监管功能,所以这里的 Port 也有相应的 MBean
[16:50:23.536]PortMBean[resin:type=Port,Server=default,port=8081,host=172.16.37.23] registered in MBeanContext[EnvironmentClassLoader$20863188[servlet-server:]]
#由于只有 pro 版本才可以使用 JNI 网络调用,所以这里抛出异常,默认的次序是首先判断可否使用
#JNI 机制,若不成,则进行对应的 JDK 网络调用
[16:50:23.597]java.lang.ClassNotFoundException: com.caucho.vfs.JniServerSocketImpl
[16:50:23.602]java.io.IOException: JNI Socket support requires Resin Professional.
[16:50:23.602]at com.caucho.vfs.QJniServerSocket.createJNI(QJniServerSocket.java)
[16:50:23.602]at com.caucho.vfs.QJniServerSocket.create(QJniServerSocket.java:75)
[16:50:23.602]at com.caucho.server.port.Port.bind(Port.java:711)
[16:50:23.602]at com.caucho.server.resin.ServletServer.bindPorts(ServletServer.java:1021)
[16:50:23.602]at com.caucho.server.resin.ServletServer.start(ServletServer.java:967)
[16:50:23.602]at com.caucho.server.deploy.DeployController.startImpl(DeployController.java:621)
[16:50:23.602]at com.caucho.server.deploy.AbstractDeployControllerStrategy.start(AbstractDeployControllerStrategy.java:56)
[16:50:23.602]at com.caucho.server.deploy.DeployController.start(DeployController.java:517)
[16:50:23.602]at com.caucho.server.resin.ResinServer.start(ResinServer.java:485)
[16:50:23.602]at com.caucho.server.resin.Resin.init(Resin.java)
[16:50:23.602]at com.caucho.server.resin.Resin.main(Resin.java:624)
[16:50:23.609]http listening to [D]_37_23_kevinxu:8081
#此 Port 用 127.0.0.1 地址开启,即只在本地开启,作用是做负载均衡分布式备份
#所有 Port 和相应属性都是在 RESIN_HOME/conf/resin.conf 中配置的
[16:50:23.610]PortMBean[resin:type=Port,Server=default,port=6803,host=127.0.0.1] registered in MBeanContext[EnvironmentClassLoader$20863188[servlet-server:]]
[16:50:23.647]java.lang.ClassNotFoundException: com.caucho.vfs.JniServerSocketImpl
#同样是 JNI 的问题
[16:50:23.648]java.io.IOException: JNI Socket support requires Resin Professional.
[16:50:23.648]at com.caucho.vfs.QJniServerSocket.createJNI(QJniServerSocket.java)
[16:50:23.648]at com.caucho.vfs.QJniServerSocket.create(QJniServerSocket.java:75)
[16:50:23.648]at com.caucho.server.port.Port.bind(Port.java:711)
[16:50:23.648]at com.caucho.server.resin.ServletServer.bindPorts(ServletServer.java:1021)
[16:50:23.648]at com.caucho.server.resin.ServletServer.start(ServletServer.java:967)
[16:50:23.648]at com.caucho.server.deploy.DeployController.startImpl(DeployController.java:621)
[16:50:23.648]at com.caucho.server.deploy.AbstractDeployControllerStrategy.start(AbstractDeployControllerStrategy.java:56)
[16:50:23.648]at com.caucho.server.deploy.DeployController.start(DeployController.java:517)
[16:50:23.648]at com.caucho.server.resin.ResinServer.start(ResinServer.java:485)
[16:50:23.648]at com.caucho.server.resin.Resin.init(Resin.java)

[16:50:23.648]at com.caucho.server.resin.Resin.main(Resin.java:624)
[16:50:23.650]hmux listening to localhost:6803

[16:50:23.651]Server[] active
#Server[]启动三阶段: initializing -> starting -> active

[16:50:23.651]HostExpandDeployGenerator[/usr/local/resin_test/hosts] starting
[16:50:23.652]HostController[] initializing
[16:50:23.709]HostMBean[resin:type=Host,Server=default,name=default] registered in
MBeanContext[EnvironmentClassLoader$20863188[servlet-server:]]

[16:50:23.709]HostController[] initialized
[16:50:23.710]HostController[] starting

[16:50:23.710]Host[] initializing
[16:50:23.711]Host[] root-directory=/usr/local/resin_test/
[16:50:23.711]MBeanServerDelegateMBean[JMImplementation:type=MBeanServerDelegate] registered in
MBeanContext[EnvironmentClassLoader$29384701[host:]]
[16:50:23.712]HostMBean[resin:type=CurrentHost] registered in
MBeanContext[EnvironmentClassLoader$29384701[host:]]
[16:50:23.759]WebAppExpandDeployGenerator[/usr/local/resin_test/webapps] redeploy false
[16:50:23.764]EarDeployGenerator[/usr/local/resin_test/deploy] redeploy false
[16:50:23.766]WebAppExpandDeployGenerator[/usr/local/resin_test/deploy] redeploy false
[16:50:23.770]WebAppExpandDeployGenerator[/usr/local/resin_test/webapps] redeploy false
[16:50:23.774]EarDeployGenerator[/usr/local/resin_test/deploy] redeploy false
[16:50:23.776]WebAppExpandDeployGenerator[/usr/local/resin_test/deploy] redeploy false
[16:50:23.782]WebAppExpandDeployGenerator[/usr/local/resin_test/webapps] redeploy false
[16:50:23.785]EarDeployGenerator[/usr/local/resin_test/deploy] redeploy false
[16:50:23.786]WebAppExpandDeployGenerator[/usr/local/resin_test/deploy] redeploy false
[16:50:23.788]resin:import '/usr/local/resin_test/host.xml' is not readable.
[16:50:23.799]Host[] initializing

[16:50:23.800]Host[] starting
[16:50:23.800]WebAppExpandDeployGenerator[/usr/local/resin_test/webapps] starting
[16:50:23.800]EarDeployGenerator[/usr/local/resin_test/deploy] starting
[16:50:23.801]WebAppExpandDeployGenerator[/usr/local/resin_test/deploy] starting
[16:50:23.801]EarDeployGenerator[/usr/local/resin_test/deploy] starting
[16:50:23.801]EarDeployGenerator[/usr/local/resin_test/deploy] starting

#==========所有的 Webapp 进入到初始化(initializing)阶段==========#
#start of ROOT, 部署一个 webapp, 当前为 ROOT
[16:50:23.803]WebAppController$24193850[] initializing
#相应的 JMX 配置
[16:50:23.805]WebAppMBean[resin:type=WebApp,Server=default,Host=default,name=/] registered in

MBeanContext[EnvironmentClassLoader$29384701[host:http://172.16.37.23:8081]]
[16:50:23.805]WebAppController$24193850[] initialized
#end of webapp

#start of webapp, 这是我部署的一个 webapp --- CustomerAdmin
[16:50:23.806]WebAppController$9717476[/CustomerAdmin] initializing
#相应的 JMX 配置
[16:50:23.807]WebAppMBean[resin:type=WebApp,Server=default,Host=default,name=/CustomerAdmin] registered in MBeanContext[EnvironmentClassLoader$29384701[host:http://172.16.37.23:8081]]
[16:50:23.807]WebAppController$9717476[/CustomerAdmin] initialized
#end of webapp

#start of webapp, licence
[16:50:23.808]WebAppController$12213370[/licence] initializing
#相应的 JMX 配置
[16:50:23.809]WebAppMBean[resin:type=WebApp,Server=default,Host=default,name=/licence] registered in MBeanContext[EnvironmentClassLoader$29384701[host:http://172.16.37.23:8081]]
[16:50:23.810]WebAppController$12213370[/licence] initialized
#end of webapp

#start of webapp, report
[16:50:23.811]WebAppController$17652030[/report] initializing
#相应的 JMX 配置
[16:50:23.811]WebAppMBean[resin:type=WebApp,Server=default,Host=default,name=/report] registered in MBeanContext[EnvironmentClassLoader$29384701[host:http://172.16.37.23:8081]]
[16:50:23.812]WebAppController$17652030[/report] initialized
#end of webapp

#==========所有的 Webapp 进入到启动(starting)阶段==========#
#starting of webapp, ROOT, 这里拿 ROOT 作加载原理分析
[16:50:23.855]WebAppController$24193850[] starting
[16:50:23.856]WebApp[http://172.16.37.23:8081] initializing
[16:50:23.856]WebApp[http://172.16.37.23:8081] root-directory=/usr/local/resin_test/webapps/ROOT
[16:50:23.856]MBeanServerDelegateMBean[JMImplementation:type=MBeanServerDelegate] registered in MBeanContext[EnvironmentClassLoader$24659469[web-app:]]
[16:50:23.857]WebAppMBean[resin:type=CurrentWebApp] registered in MBeanContext[EnvironmentClassLoader$24659469[web-app:]]
[16:50:23.960]servlet-mapping *.jsp -> resin-jsp
[16:50:23.960]servlet-mapping *.jspx -> resin-jspx
[16:50:23.961]servlet-mapping *.xtp -> resin-xtp
[16:50:23.961]servlet-mapping / -> resin-file
[16:50:23.999]resin:import '/usr/local/resin_test/webapps/ROOT/WEB-INF/web.xml'
#servlet 映射,这里的 db_proxy 由于是引用的 jar 里的一个 Servlet

#所以紧接下来并没有装载预编译类的过程,这个看下一个 webapp 加载示范
[16:50:24.048]servlet-mapping /db_proxy -> db_proxy
#没有找到 resin 特定的 resin-web.xml 配置文件
[16:50:24.049]resin:import '/usr/local/resin_test/webapps/ROOT/WEB-INF/resin-web.xml' is not readable.
[16:50:24.049]WebApp[http://172.16.37.23:8081] initializing
#之后发现每个 webapp 的 EnvironmentClassLoader 的 id 都不同,说明每个 webapp 分别对应一个
#自己的 EnvironmentClassLoader,但是所有 webapp 的父 EnvironmentClassLoader 都为同一个,可以
#看到之后每个 webpp 的 parent:EnvironmentClassLoader 的 id 都相同,这里为 29384701
[16:50:24.055]creating JNDI java: model for EnvironmentClassLoader$24659469[web-app:http://172.16.37.23:8081] parent:EnvironmentClassLoader$29384701[host:http://172.16.37.23:8081]
[16:50:24.055]WebApp[http://172.16.37.23:8081] initialized
[16:50:24.056]WebApp[http://172.16.37.23:8081] starting

[16:50:24.058]Servlet[resin-jsp] starting
[16:50:24.070]resin-jsp init
[16:50:24.079]Servlet[resin-jsp] started
[16:50:24.079]Servlet[resin-jspx] starting
[16:50:24.080]resin-jspx init
[16:50:24.080]Servlet[resin-jspx] started
[16:50:24.081]WebApp[http://172.16.37.23:8081] active
[16:50:24.098]WebAppController$24193850[] active

#starting of webapp, CustomerAdmin
[16:50:24.099]WebAppController$9717476[/CustomerAdmin] starting
[16:50:24.100]WebApp[http://172.16.37.23:8081/CustomerAdmin] initializing
[16:50:24.100]WebApp[http://172.16.37.23:8081/CustomerAdmin] root-directory=/usr/local/resin_test/webapps/CustomerAdmin
[16:50:24.101]MBeanServerDelegateMBean[JMImplementation:type=MBeanServerDelegate] registered in MBeanContext[EnvironmentClassLoader$28708894[web-app:/CustomerAdmin]]
[16:50:24.101]WebAppMBean[resin:type=CurrentWebApp] registered in MBeanContext[EnvironmentClassLoader$28708894[web-app:/CustomerAdmin]]
[16:50:24.141]servlet-mapping *.jsp -> resin-jsp
[16:50:24.142]servlet-mapping *.jspx -> resin-jspx
[16:50:24.143]servlet-mapping *.xtp -> resin-xtp
[16:50:24.143]servlet-mapping / -> resin-file
[16:50:24.163]resin:import '/usr/local/resin_test/webapps/CustomerAdmin/WEB-INF/web.xml'
[16:50:24.181]In-place class redefinition (HotSwap) is not available. In-place class reloading during development requires a compatible JDK and -Xdebug.
#start filter 映射, EncodingFilter
[16:50:24.184]loading pre-compiled class com.qq.customeradmin.web.filter.EncodingFilter from /usr/local/resin_test/webapps/CustomerAdmin/WEB-

INF/classes/com/qq/customeradmin/web/filter/EncodingFilter.class
[16:50:24.191]filter-mapping * -> EncodingFilter
[16:50:24.192]filter-mapping * -> EncodingFilter
#end filter 映射

#start filter 映射, AuthFilter
[16:50:24.195]loading pre-compiled class com.qq.customeradmin.web.filter.AuthFilter from /usr/local/resin_test/webapps/CustomerAdmin/WEB-INF/classes/com/qq/customeradmin/web/filter/AuthFilter.class
[16:50:24.197]filter-mapping * -> AuthFilter
[16:50:24.197]filter-mapping * -> AuthFilter
#end filter 映射

#start servlet 映射, ActionGateway
[16:50:24.203]loading pre-compiled class com.qq.customeradmin.web.servlet.ActionGateway from /usr/local/resin_test/webapps/CustomerAdmin/WEB-INF/classes/com/qq/customeradmin/web/servlet/ActionGateway.class
#把相关的类也装载进来
[16:50:24.212]loading pre-compiled class com.qq.customeradmin.common.web.event.ActionHandler from /usr/local/resin_test/webapps/CustomerAdmin/WEB-INF/classes/com/qq/customeradmin/common/web/event/ActionHandler.class

[16:50:24.215]servlet-mapping ag -> ActionGateway
#end servlet 映射

[16:50:24.216]resin:import '/usr/local/resin_test/webapps/CustomerAdmin/WEB-INF/resin-web.xml' is not readable.

[16:50:24.217]WebApp[http://172.16.37.23:8081/CustomerAdmin] initializing
[16:50:24.225]creating JNDI java: model for EnvironmentClassLoader$28708894[web-app:http://172.16.37.23:8081/CustomerAdmin] parent:EnvironmentClassLoader$29384701[host:http://172.16.37.23:8081]
[16:50:24.226]WebApp[http://172.16.37.23:8081/CustomerAdmin] initialized
[16:50:24.226]WebApp[http://172.16.37.23:8081/CustomerAdmin] starting
[16:50:24.227]Servlet[resin-jsp] starting
[16:50:24.228]resin-jsp init
[16:50:24.229]Servlet[resin-jsp] started
[16:50:24.229]Servlet[resin-jspx] starting
[16:50:24.231]resin-jspx init
[16:50:24.232]Servlet[resin-jspx] started
[16:50:24.234]WebApp[http://172.16.37.23:8081/CustomerAdmin] active
[16:50:24.237]WebAppController$9717476[/CustomerAdmin] active
... ...
[16:50:24.375]Host[] active

[16:50:24.375]HostController[] active
[16:50:24.376]ServerController$5924809[] active
[16:50:24.377]Resin started in 1530ms
#Humx 主要是做集群通信的
[16:50:24.499]java.lang.ClassNotFoundException: com.caucho.server.hmux.HmuxClusterRequest
[16:50:24.546]java.lang.ClassNotFoundException: com.caucho.server.hmux.HmuxClusterRequest
[16:50:24.576]java.lang.ClassNotFoundException: com.caucho.server.hmux.HmuxClusterRequest
[16:50:24.606]java.lang.ClassNotFoundException: com.caucho.server.hmux.HmuxClusterRequest
[16:50:24.636]java.lang.ClassNotFoundException: com.caucho.server.hmux.HmuxClusterRequest

总结分析:

[Resin]对应多个[ServerController],[ServerController]对应多个[Host],每个[Host]关联一个[HostController], 每个[Host]对应多个 WebApp,每个 WebApp 关联一个 WebappController.
以上元素都是走 initializing, starting, active 流程.主要是因为它们都是用 Lifecycle 作为生命周期控制对象.所以状态是统一的.

运行时调试日志

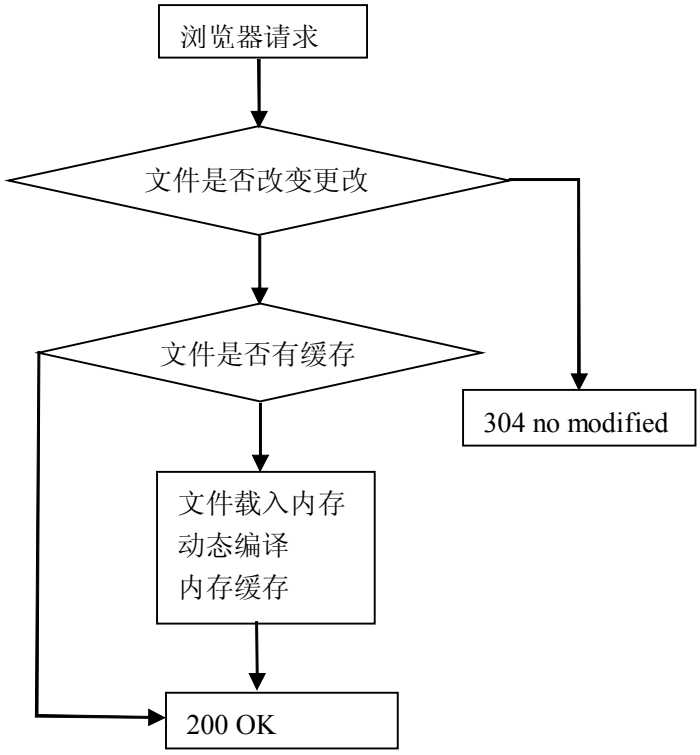
下面是接下来一段运行时的 debug.log:
#开启连接
[19:30:39.772]starting connection TcpConnection[id=resin-tcp-connection-172.16.37.23:8081-2,socket=QSocketWrapper[Socket[addr=/10.2.70.51,port=3400,localport=8081]],port=Port[172.16.37.23:8081]], total=5
#浏览器发来的请求头,这里默认会返回主页 index.jsp,注意,这里是 jsp 页面
[19:30:39.774][0] GET / HTTP/1.1
[19:30:39.774][0] Remote-IP: 10.2.70.51:3400
[19:30:39.774][0] Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/x-shockwave-flash, */*
[19:30:39.774][0] Accept-Language: zh-cn
[19:30:39.775][0] Accept-Encoding: gzip, deflate
[19:30:39.775][0] User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)
[19:30:39.775][0] Host: 172.16.37.23:8081
[19:30:39.775][0] Connection: Keep-Alive

#由于是初次加载,所以需要载入内存并进行动态编译
[19:30:39.786]invoke (uri:/index.jsp -> resin-jsp)
[19:30:39.793]uri:/index.jsp(cp:,app:/usr/local/resin_test/webapps/ROOT) -> /usr/local/resin_test/webapps/ROOT/index.jsp

[19:30:39.830]loading pre-compiled page for /index.jsp
#成功响应
[19:30:39.875][0] HTTP/1.1 200 OK
[19:30:39.875][0] Content-Type: text/html
[19:30:39.876][0] Transfer-Encoding: chunked
[19:30:39.876][0] write-chunk(7)
[19:30:39.877][0] keepalive
#关闭连接
[19:31:44.881]closing connection TcpConnection[id=resin-tcp-connection-172.16.37.23:8081-2,socket=QSocketWrapper[Socket[addr=/10.2.70.51,port=3400,localport=8081]],port=Port[172.16.37.23:8081]], total=6

总结分析:

显然 Resin 是 lazy-loading,包括延迟载入内存,延迟动态编译.



Watchdog 机制

Resin 中实现了一个叫做 Watchdog 监控机制.他本身类似一个守护进程, 作用是监控 resin 的运行,并在意外时 restart Resin 服务主进程.Resin 3.0.x 是用 perl(也就是 wrapper.pl)中实现 watchdog 监控功能的.我们服务器目前 resin 的版本大多还是 3.0.xx 在 3.1.x 中，实现了一个内置的 watchdog 进程来进行监控.

启动和关闭 watchdog(perl)进程(Resin 3.0.x)

e.g. 启动 http.sh 脚本即可, 内部默认调用了 wrapper.pl(wrapper.pl 即为 watchdog 实现脚本).

启动和关闭 watchdog(Java)进程(Resin 3.1.x)

e.g.启动 resin: linux> java -jar /usr/local/share/resin/lib/resin.jar -conf /etc/resin/resin.conf start

Resin 实际是通过"start", "stop", and "restart"参数来启动 resin.jar."start"和"stop"发送消息给一个 watchdog 进程，watchdog 进程负责开启真实的 Resin 进程.watchdog 监视 Resin 的状态，并在必要时重启 resin 以提高可靠性.

在有 watchdog manager 之前,假如你需要 kill Resin, 你首先需要 kill watchdog 进程. 否则如果你仅仅只 kill 了 Resin 进程,watchdog 将会自动 restart 它. JDK 1.5 中包含一个叫做"jps"的命令工具可以查看所有的 java 进程的 pid. WatchdogManager 为 watchdog 实例，Resin 为 Resin 实例.

网络模型分析

首先来个玩笑:-).

隐喻：

故事场景: Resin 地区

Port 8080: 为 Rein 地区的一个传统服装贸易市场.
TcpConnection: 这里有很多服装公司如 Jack Jones, isprit.
ThreadPool: 为 Resin 一家大型的服装加工厂.

Resin 地区开办了一个大型的服装贸易市场 Resin Port 8080，作为一个大型服装贸易市场,自然会吸引大量客户来 Resin Port 8080 市场寻觅商家，我们的大 Port 8080 市场开放后，弄了一拨商家（子线程）来为客户服务，这些商家(TcpConnection)互相竞争(accept)，获取和客户进行贸易的机会。如果客户搞头大，合作周期长，那么商家就会和此客户保持一段合作时间(Keepalive)。商家们拿到客户单(Request)之后马上会找加工厂授权加工产品(这里加工厂为 ThreadPool)由于 SUN 公司之前开办 ThreadPool 加工厂,所以 Resin 地区就开了一家自个的服装加工厂(ThreadPool).

早期,8080 市场开张时放入了_minSpareConnection(Port 中的字段)个商家, 相应的 ThreadPool 加工厂也只有_threadMinIdleMin 个工人. 当然, 这也是市场开放必须具备的最小规模,
最早期,客户不多,这些商家足够应付. 随着贸易增长,客户增加, 市场规模扩大, 商家也越来越多,ThreadPool 加工厂的工人也越招越多. 但是,市场在各方面也有自己的限制因素,比如场地,地区的资源等. 所以 Resin 地区强行规定商家数

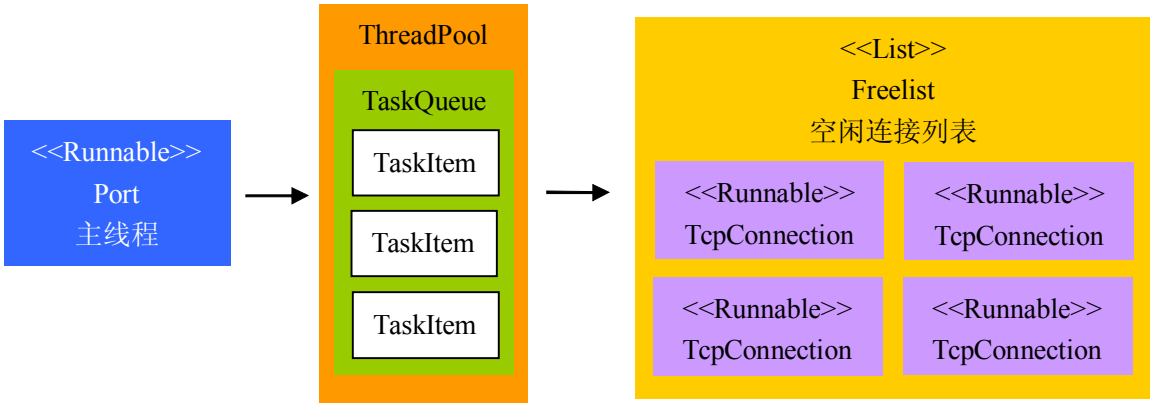
量不能超过_connectionMax 个 (_connectionMax 为 Port 字段,写死为 512),当然,有关系的商家也可以走后门超过这个限制(修改 Rein 源代码后重新编译), 加工厂也受一些资源限制,规定加工厂工作数量不能超过 8192 人,当然,这个公司当然比政府要灵活多了,可以根据市场状况随机更改(在 resin.conf 中配置).

其实只要市场政策开放,这些限制都可以突破(resion.conf 提供所有必要的配置项,在 resin.conf 中可以对市场进行宏观调控).

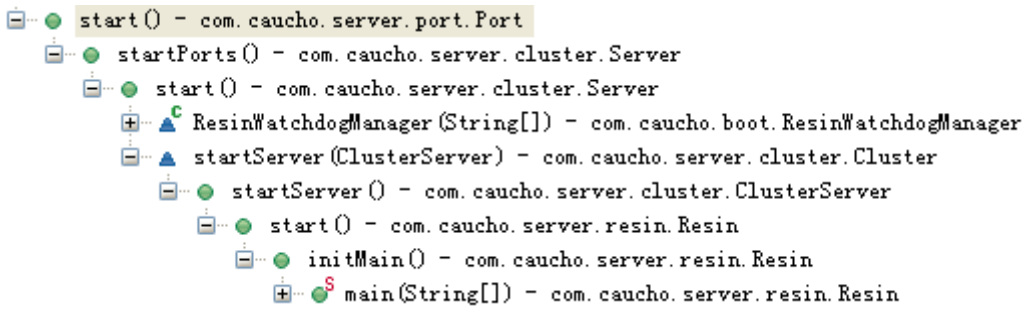
当 Resin Port 8080 市场达到饱和之后, 本地发展受限, Port 8080 便到其它地区如 Resin X 开办了相同的贸易市场 ReinX 8080. 这时, 原来在 Resin Port 8080 的商家,如 Jack Jones, 可以在 Resin X Port8080 办分公司(也就是 resin 中的集群方式), 也就是扩大市场以容纳更多的客户(对应 resin 中用 Session 复制,负载均衡等手段).

模型分析

网络模型为：线程池&子线程竞争 accept.



threadCount connectionCount



Port 所处于的线程为父，做一个主循环, 负责创建子连接。目前采用的 I/O 机制为最普通的 SocketServer 阻塞式 accept. JNI 方式的 epoll/select 调用只在 professional 版本中支持。

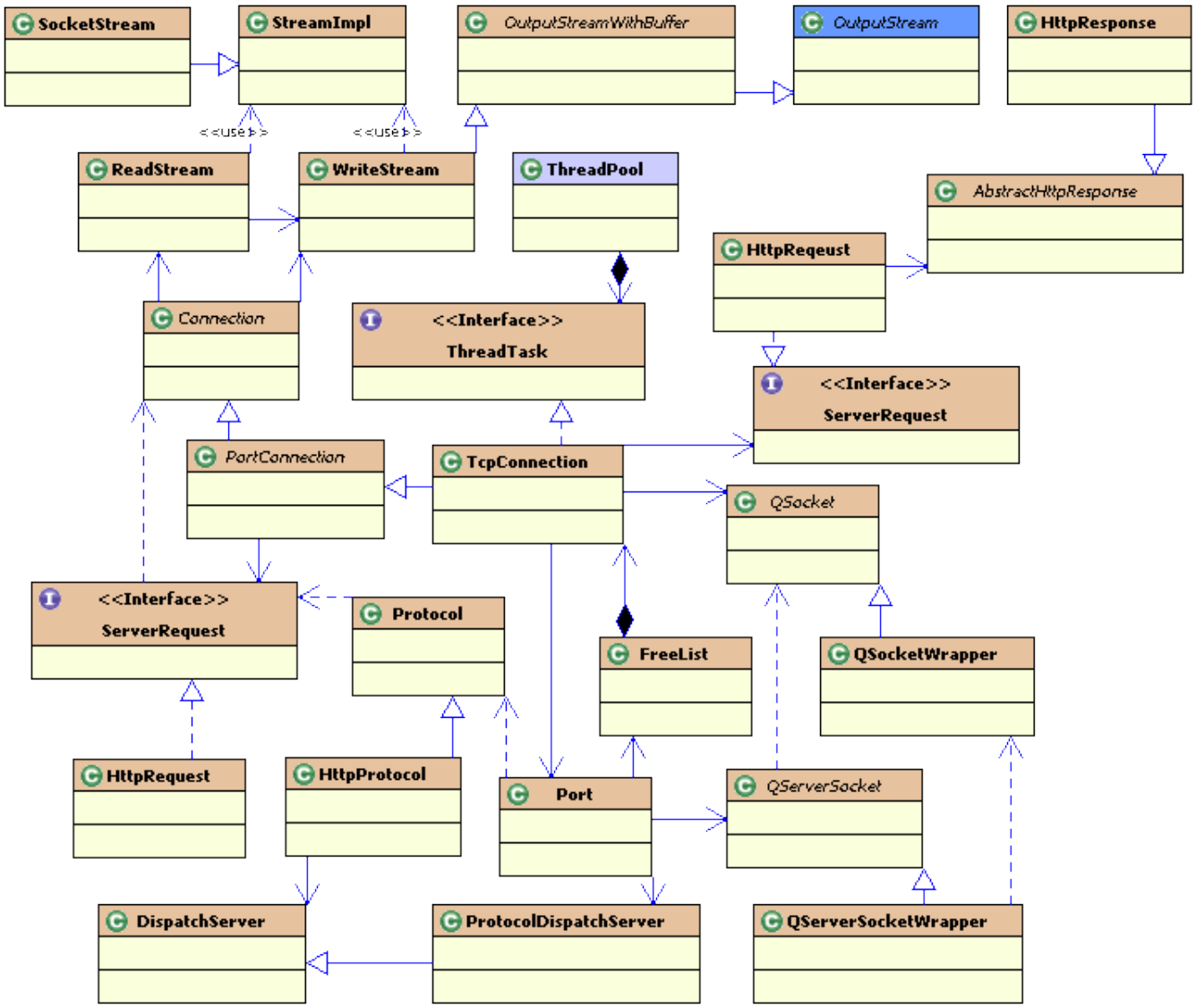
启动时: 无连接句柄, TcpConnection 对象少,但至少要_acceptThreadMin 个空闲 TcpConnection 来运作. ThreadPool 工作者总量也少.

繁忙时: 连接多(但最多不能超过_connectionMax 个 TcpConnection 对象. ThreadPool 中工作者不能多于 maxThreadCount 个.

空闲时: 连接句柄少,很多竞争 accept 的 TcpConnection 发现无事可做,就自行 free(),free()过程是先看看 FreeList 里面有没有地址收容它,Freelist 默认只给 32 个空缺(hard coding, 政策很死啊), 如果有收容之地,那么就扔进去, 如果没有, 就 bytebyte 了,丢给 gc 收集去. 线程池这边也是大裁员,如果好久都认领不到任务的线程就只好被解雇,3166 啦.

繁忙,空闲,就这么周而复始.

特点(Feature): 可升缩性, I/O 句柄(连接) & 工作者线程 unbind 关系.



Port 为一个端口的占有者，比如默认的 8080，它负责执行主循环逻辑，建立连接对象。
TcpConnection 代表一个连接实体,在某一时刻,一个 TcpConnection 要么对应至多一个 I/O 句柄并且在处理任务,要么还处于竞争 acceipt 装载.一个 TcpConnection(实现 Runnable 接口)并不是一个主动对象,事实上,TcpConnection 需要 TaskItem 作为主动对象来启动 TcpConnection 的工作, 关于细节后续详谈.
QSocket 是个抽象类，抽象了型别(Socket 或者 JNI Socket,但 JNI Socket 只可以在 Resin Pro 版本中使用)。
QSocketWrapper 是一个具体 QSocket 类,包装了 java.net.Socket 对象.
QServerSocket 是个抽象类,职责是构造具体的 QSocket 对象.
QServerSocketWrapper 是一个具体的 QServerSocket 类.

源码剖析：

Port

首先是 Port 的主执行流程 Run().这里需要先介绍一下 ThreadPool,ThreadPool 是一个全局的线程池,基本所有的任务调度(包括 Port 在内)都交给 ThreadPool 进行处理.

隐喻: Port 8080 进行市场内的监管,调控工作.ThreadPool 为一家加工厂,但不专职为 Port8080 市场作服装加工,包括很多其它的业务, 也就是 Port 8080(TcpConnection)并不完全占据 ThreadPool.

流程概述:

WHILE(true)//循环体
Thread.yield();　　Thread.sleep(10);//做的个 XX 优化
线程条件：当前 正在开启的线程+空闲线程<可开启的最小线程数.
连接条件：当前开启的连接小于最大连接
如果不满足条件，则线程等待.这里设置成等待 60 秒.
否则在空闲连接列表中拿连接，这里有两种情况：
* 一是空闲连接未用完，则取出,如果取出的连接还未初始化，则初始化一个新的 TcpConnection
* 二是空闲连接用光了，则动态分配一个来处理请求
最后在 ThreadPool 中取线程(ThreadTask)来执行连接。
END WHILE

```
/**
 * port线程负责创建新连接
 */
public void run() {
    while (!_lifecycle.isDestroyed()) {
        boolean isStart;
        try {
/**
 * 使当前正在执行的线程对象暂停,并允许其他线程执行 need delay to avoid spawing too many
 * threads over a short time, when the load doesn't justify it
 */
            Thread.yield();
            Thread.sleep(10);
            synchronized (this) {
/**
 * 正在开启的线程数+空闲线程数<可接收的最小线程,那么isStart=true;
 * 这里开启的线程数 意思是分配给了TcpConnection,但是在作accept之前的TcpConnection的数量,
 * 在accept之前
 */
            isStart = _startThreadCount + _idleThreadCount < _acceptThreadMin;
/**
```

```
 * 如果最大连接数<=当前连接数,isStart=false
 */
          if (_connectionMax <= _connectionCount)
              isStart = false;
              if (!isStart) {
                  Thread.interrupted();
                  wait(60000);
              }

/**
 * 如果决定start,那么连接数和开启线程数都增加,这是自然
 */
              if (isStart) {
                  _connectionCount++;
                  _startThreadCount++;
              }
          }
/**
 * 用lifecycle来标示生命周期状态,Resin里面很多组件的状态是比较统一的,都用的Lifecycle来标示
 * 一个组件的生命周期
 */
          if (isStart && _lifecycle.isActive()) {
/**
 * 看freeList中拿连接，这里有两种情况：
 * 一是空闲连接未用完，则取出,如果取出的连接还未初始化，则初始化一个新的TcpConnection
 * 二是空闲连接用光了，则动态分配一个来处理请求
 */
              TcpConnection conn = _freeConn.allocate();
              if (conn == null) {
                  conn = new TcpConnection(this, _serverSocket.createSocket());
                  conn.setRequest(_protocol.createRequest(conn));
          }
              conn.start();//done
              //从这里入口来处理连接
              ThreadPool.getThreadPool().schedule(conn);
          }
       } catch (Throwable e) {
          e.printStackTrace();
       }
    }
}
```

TcpConnection

主要介绍 TcpConnection 的核心执行体 run(),它的主要工作就是竞争 accept 响应请求,并且 Http 的 keepalive 特性也是由它负责.
但是如果在处理中长时间无事可做,则会销毁自身.

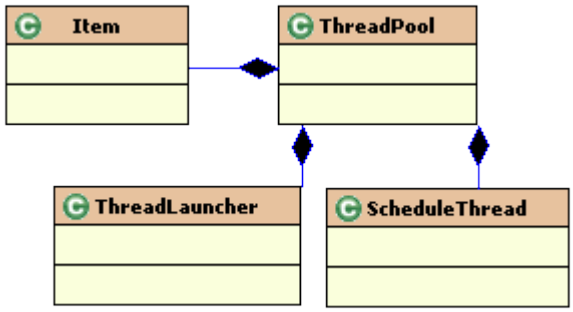
```
/**
 * Runs as a task.
 */
public void run() {
    Port port = getPort();
    /**
     * 看当前的connection是否仍是Keepalive的
     */
    boolean isKeepalive = _isKeepalive;
    _isKeepalive = false;
    boolean isFirst = !isKeepalive;
    ServerRequest request = getRequest();// 之前协议分析器已经解析出request并set给了此TcpConnection
/**
 * 看此请求是否等待读，如果是HttpRequest,总是返回true
 */
    boolean isWaitForRead = request.isWaitForRead();
    // 把当前线程的名字给改掉,调试日志中打印出的resin-tcp-connection-xxx字样
    Thread thread = Thread.currentThread();
    String oldThreadName = thread.getName();
    thread.setName(_id);
    if (isKeepalive) {
        port.keepaliveEnd(this);
    }
/**
 * 标记一下一个新的线程正在运行 port._threadCount++
 */
    port.threadBegin(this);// done

    ClassLoader systemLoader = ClassLoader.getSystemClassLoader();

    long startTime = Alarm.getExactTime();
    // 把系统ClassLoader设置为此线程的环境ClassLoader
    thread.setContextClassLoader(systemLoader);
    // _admin.register();
    try {
        // 此连接的_thread为当前线程
        _thread = thread;
        // TODO: _isDead是由谁控制
        while (!_isDead) {
```

```
            // 如果isKeepalive,则不需要再accept,下次继续用此Socket服务
            if (isKeepalive) {
            } else if (!port.accept(this, isFirst)) {// 初始化一个新的Socket,赋予给this
                return;
            }
            isFirst = false;
            try {
                thread.interrupted();
                // clear the interrupted flag
                do {
                    thread.setContextClassLoader(systemLoader);
                    isKeepalive = false;
/**
 * isWaitForRead 由HttpRequest返回的总是true,
 *
 * 如果port没有关闭,并且有可读数据
 */
                    if (!port.isClosed()
                            && (!isWaitForRead || getReadStream()
                                    .waitForRead())) {
                        synchronized (_requestLock) {
/*
 * 如果是keepalive请求,则返回true
 */
                            isKeepalive = request.handleRequest();
                        }
                    }
                } while (isKeepalive && waitForKeepalive()
                        && !port.isClosed());
/**
 * 如果是keepalive连接,就暂时return,之后可被重新调用
*/
                if (isKeepalive) {
                    return;
                } else {
/**
 * 如果不是keepalive连接,就关闭request,protocolCloseEvent在这里其实是空方法,HTTP协议没有什
 * 么善后工作
*/
                    getRequest().protocolCloseEvent();
                }
            } catch (ClientDisconnectException e) {
                isKeepalive = false;
                if (log.isLoggable(Level.FINER))
```

```
                log.finer("[" + getId() + "] " + e);
            } catch (IOException e) {
                isKeepalive = false;
                if (log.isLoggable(Level.FINE))
                    log.log(Level.FINE, "[" + getId() + "] " + e, e);
            } finally {
                thread.setContextClassLoader(systemLoader);
                if (!isKeepalive)
                    closeImpl();
            }
        }
    } catch (Throwable e) {
        log.log(Level.WARNING, e.toString(), e);
        isKeepalive = false;
    } finally {
        thread.setContextClassLoader(systemLoader);
        // _admin.unregister();
        port.threadEnd(this);
        if (isKeepalive)
            keepalive();
        else
            //在执行完毕后free,如果不需要销毁,则可以放回Freelist,若Freelist已满,则销毁
            free();
        _thread = null;
        thread.setName(oldThreadName);
    }
}
```

线程池模型

resin 中的线程池也是自行实现的，并非使用 JDK 中提供的。



ThreadPool 为外界访问的入口,并且 ThreadPool 为 Singleton.ThreadPool 在实例化时同时构造了一个 ThreadLauncher 还有一个 ScheduleThread 对象.ThreadPool 向外暴露的 Schedule 方法是用来调度任务的,任务必须是 Runnable 类型的,这里典型就是我们的 TcpConnection.schedule 方法有两个调用环境,一个是外部显示调用,另一种则是线程调度器(ScheduleThread)周期性调用.

ThreadPool 的关键字段:

ArrayList<Item> _threads　　//工作者线程队列

ArrayList<Runnable> _taskQueue　　//任务队列

ArrayList<ClassLoader> _loaderQueue //ClassLoader 队列,这个是和任务队列对应的,任务执行的环境

private Item _idleHead; //指向一个工作者对象,这是一个细节之源

ThreadPool 的任务调度有一些细节,之后详细讨论:

ThreadLauncher 也是一个循环线程,工作是创建工作者线程对象(Item). 并一直循环运行 Item 的创建工作. 它在 ThreadPool 实例化时启动.

ScheduleThread 是一个循环线程.工作是调度任务,即分配任务到指定工作者线程.线程调度器(ScheduleThread)周期性调度.线程调度器循环检查任务队列, 如果任务队列有待处理任务,则拿出任务来调度(ThreadPool.schedule),否则就 wait 1 分钟(中途来了任务可以把它唤醒).

原理分析:

Schedule-Workder 调度模式, 即调度者负责分配任务到工作者线程.

在启动时: LauncherThread 只需分配 threadMin 个工作者线程.任务调度器 ScheduleThread 定期检查任务,如果没有任务就 wait 一会(1 分钟).

在繁忙时: 任务多了, ThreadLauncher 发现各个工作者线程都很忙,就不断分配新的工作者线程(但数量 threadCount 小于上限 threadMax, threadMax 默认为 8192 个).

在空闲时: ThreadLauncher 也无须分配工作线程, 调度器也无任务可调度, 工作者自然亦无事可做,工作者如果在一段时间内总找不到活干,它就退休了(该工作者线程销毁了).

源码剖析

ThreadLauncher

首先介绍 ThreadLauncher，我们可以在这里把 ThreadLauncher 比喻成一个公司的 HR，
下面是它的 run()方法:

```
/**
         * 隐喻：这家伙的职业生涯开始，也就是公司刚成立时先招满最低限度的threadIdleMin个工人，招聘计划刻不容缓，所以招聘过程中
         * 不能片刻等待，所以传入wait时间值为0．基本人数招满后就轻松许多了．
         * 公司正常运作．大HR就猫在这一个大while循环的定期招工人，这会可以招招停停，这里给它招工一个，可以等10秒后再招
         * 并且招完一个，马山yield让出CPU让别的线程去工作．
         */
public void run() {
    ClassLoader systemLoader = ClassLoader.getSystemClassLoader();
    Thread.currentThread().setContextClassLoader(systemLoader);
    // 先创建最小限度的Idle线程，开启它们
    try {
        for (int i = 0; i < _threadIdleMin; i++)
            startConnection(0);
    } catch (Throwable e) {
        e.printStackTrace();
    }
    while (true) {
        try {
            startConnection(10000);
            // Thread.currentThread().sleep(5);
            // 使自己暂停，让别的线程去运行．就是主动让出时间片的意思
            Thread.currentThread().yield();
        } catch (OutOfMemoryError e) {
            System.exit(10);
        } catch (Throwable e) {
            e.printStackTrace();
        }
    }
}
```

ThreadLauncher 的第二个关键方法就是 startConnection，它的主要工作是新建工人:

```
/**
 * Starts a new connection 它等waitTime的时间，然后看能不能开启新的工作线程，隐喻：
 * 就是招聘一个工人，这个waitTime表示Launcher这家伙招聘一个工人之后可以等多久再招
```

```
 */
private boolean startConnection(long waitTime)
        throws InterruptedException {
    boolean doStart = true;
    synchronized (_idleLock) {
        int idleCount = _idleCount;

        /**
         * 如果线程最大限度小于当前线程数+开启的线程数量
         * 隐喻: _threadMax代表公司能容纳的最大员工限额,_threadCount为当前员工人
         * 数,_startCount为实习生人数. 简单的说就是:如果正式员工加上试用工超过了最大员工限额,就
         * 不要招人了,doStart为指令
         */
        if (_threadMax < _threadCount + _startCount) {
            doStart = false;
            /**
             * 如果最小限度的空闲线程数小于空闲数量+开启的线程数量 隐喻:
             * 如果闲人和试用工超过了公司可容纳的最低闲人限度,那么也不 招,也难怪,公司有那么多闲人和
             * 新人，还招人干嘛
             */
        } else if (_threadIdleMin < idleCount + _startCount) {
            doStart = false;
        }
        /**
         * 如果确定启动,那么startCount自增 隐喻: 如果决定招人,那么试用工人数增加
         */
        if (doStart) {
            _startCount++;
        }
    }
    if (doStart) {
        try {
            /**
             * 是不是奇怪在这里,Item创建了并没有被加入到线程队列,其实是Item在它的run开始时自己把
             * 自己加入到
             * 工作线程队列的 隐喻:新来的员工Item,开始正式干活了
             */
            Item poolItem = new Item();
            Thread thread = new Thread(poolItem, poolItem.getName());
            thread.setDaemon(true);
            thread.start();
        } catch (Throwable e) {
            // 开启后,把 正在开启的线程数量 减少
            /**
```

```
                 * 隐喻：意外发生,比如毁约了.要把先前加入的试用工人数减一
                 */
                _startCount--;//
                e.printStackTrace();
                if (_startCount < 0) {
                    Thread.dumpStack();
                    _startCount = 0;
                }
            }
        } else {
            /**
             * 隐喻:如果不计划招人,那就等些时日
             */
            Thread.interrupted();
            synchronized (this) {
                wait(waitTime);
                return false;
            }
        }
        return true;
    }
```

ScheduleThread

以下是 ScheduleThread 的 run()方法:

```
public void run() {
/**
 * 设置线程执行上下文为系统类装载器
 */
    ClassLoader systemLoader = ClassLoader.getSystemClassLoader();
    Thread thread = Thread.currentThread();
    thread.setContextClassLoader(systemLoader);
/**
 * 主循环,负责任务调度工作
 */
    while (true) {
        try {
            Runnable task = null;
            ClassLoader loader = null;
            Thread.interrupted();
            synchronized (_taskQueue) {
                if (_taskQueue.size() > 0) {
```

```
                System.out.println("Fetching from taskQueue");
                task = _taskQueue.remove(0);
                loader = _loaderQueue.remove(0);
            } else {
                try {
                    _taskQueue.wait(60000);
                } catch (Throwable e) {
                    thread.interrupted();
                    log.finer(e.toString());
                }
            }
        }
        if (task != null) {
            schedule(task, loader, _threadIdleMin, MAX_EXPIRE,false);
        }
    } catch (OutOfMemoryError e) {
        System.exit(10);
    } catch (Throwable e) {
        e.printStackTrace();
    }
  }
}
```

com.caucho.util.ThreadPool$Item

Item(工作者线程)的 run 方法,主要负责具体执行任务:

```
public void run() {
    _thread = Thread.currentThread();
    synchronized (_idleLock) {
        _threadCount++;
        // 在启动中的线程减一,因为这个家伙已经到运行状态了
        // 隐喻: 从试用工转正了,转正后正式加入正式工列表(_threads)
        _startCount--;
        _threads.add(this);
        if (_startCount < 0) {
            System.out.println("ThreadPool start count is negative: "
                    + _startCount);
            _startCount = 0;
        }
    }
    try {
        // 隐喻: 既然是正式工了,就要开始干活了
        runTasks();
```

```
    } finally {
        synchronized (_idleLock) {
            // 被解雇后,自然正式工人数减少
            _threadCount--;
            // 并从正式工列表中删除
            _threads.remove(this);
        }
        if (_threadCount < _threadIdleMin) {
            synchronized (_launcher) {
                _launcher.notifyAll();
            }
        }
    }
}
```

Item 的关键方法之二 --- runTasks:

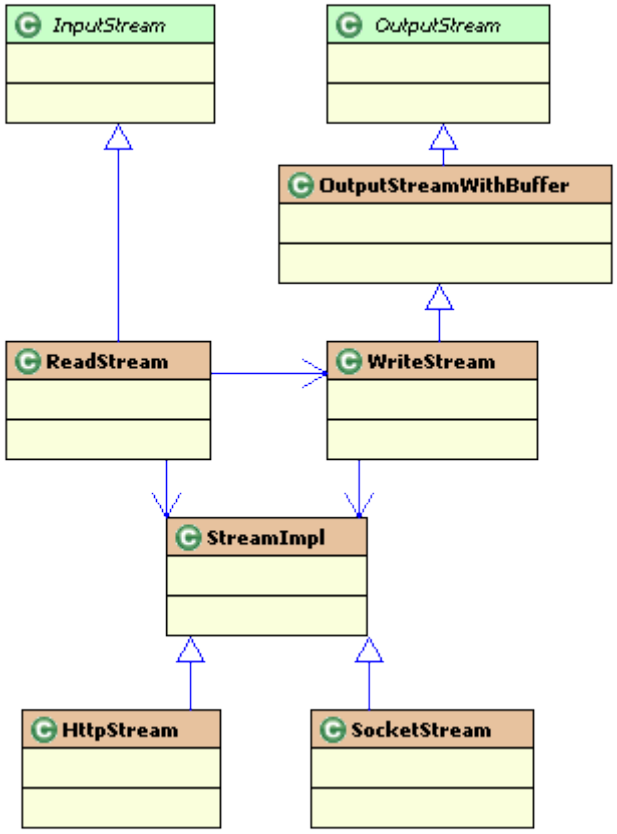
```
/**
    * 主要工作,运行所有的任务 隐喻: 此工人定期(while循环)到队列里查看是否有任务要做,在检查有没有
    * 任务做之前,它先看看空闲的人数,如果发现当前的闲人比允许的闲人上限还多,那肯定是没有事情做了,趁
    * 早return回家休息,之后再进入下一天的工作(下一次while循环),如果等很久都没有工作可做,说明公司
    * 闲人太多,得裁员了
*/
private void runTasks() {
    _threadResetCount = _resetCount;
    Thread thread = Thread.currentThread();
    ClassLoader systemClassLoader = ClassLoader.getSystemClassLoader();
    boolean isIdle = false;
    while (true) {
        try {
            // put the thread into the idle ring
            /**
             * 隐喻: 这里绕了道弯弯, 其实意思就是第一次执行,无论如何如何isIdle都为false
             * 所以,第一次while循环肯定会先把自己放到idle链表里.
             */
            if (!isIdle) {
                _isQueuePriority = true;
                isIdle = true;
                synchronized (_idleLock) {
                    // 隐喻: 如果当前空闲人数超过了最大闲人限度,嘿嘿,回家happy去,明天再来
                    if (_threadIdleMax < _idleCount) {
                        return;
                    }
```

```
            _next = _idleHead;
            _prev = null;
            _isIdle = true;
            if (_idleHead != null)
                _idleHead._prev = this;
            _idleHead = this;
            _idleCount++;
            if (_scheduleWaitCount > 0)
                _idleLock.notifyAll();
        }
    }
    Runnable task = null;
    ClassLoader classLoader = null;
    // clear interrupted flag
    Thread.interrupted();
    // wait for the next available task
    synchronized (this) {
        // 隐喻:如果没有工作做,那么就等会,估计60秒钟以内肯定有任务到手
        if (_task == null) {
            thread.setContextClassLoader(systemClassLoader);
            wait(60000L);
        }
        task = _task;
        _task = null;
        classLoader = _classLoader;
        _classLoader = null;
    }
    // if the task is available, run it in the proper context
    /**
     * 隐喻:如果任务到手就执行任务,就在适当的环境中执行它
     */
    if (task != null) {
        isIdle = false;
        thread.setContextClassLoader(classLoader);
        try {
            System.out.println(this.getId()+" :     runtask");
            task.run();
        } catch (Throwable e) {
        log.log(Level.WARNING, e.toString(), e);
        } finally {
            thread.setContextClassLoader(ClassLoader
                    .getSystemClassLoader());
        }
    } else {
```

```
                // 如果还没有任务
                boolean isDead = false;
                boolean isReset = false;
                // check to see if we're over the idle thread limit
                synchronized (_idleLock) {
                // 如果空闲线程数最大限度<当前空闲数,或者 TODO 后面这个_resetCount尚不清楚啥意思
                    if (_isIdle
                                && (_threadIdleMax < _idleCount || _resetCount !=
                _threadResetCount)) {
                        // 这个标志一标, 就得被解雇了, 公司这么清闲,留这里白混啊
                        isDead = true;
                        isReset = _resetCount != _threadResetCount;
                        Item next = _next;
                        Item prev = _prev;
                        _next = null;
                        _prev = null;
                        _isIdle = false;
                        if (next != null)
                            next._prev = prev;
                        if (prev != null)
                            prev._next = next;
                        else
                            _idleHead = next;
                        _idleCount--;
                    }
                }
                if (isReset) {
                    synchronized (_launcher) {
                        _launcher.notifyAll();
                    }
                }
                if (isDead) {
                    // 被解雇了
                    return;
                }
            }
        } catch (Throwable e) {}
    }
}
```

I/O 模型

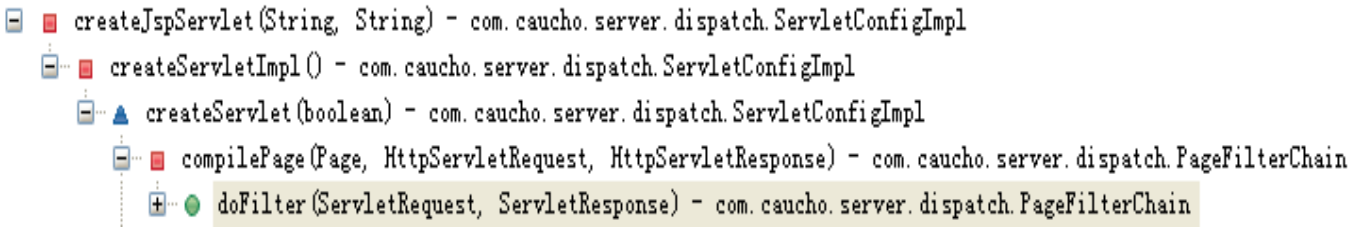
Resin 简单封装了一个缓冲式读写 I/O：



I/O 用的代理模式手法，ReadStream， WriteStream 为读写流的代理, 实现成缓冲式机制. ReadStream,WriteStream 主要作用是封装各种 StreamImpl 的差异，对外提供一致的接口,使得所有的流操作都共用这一套 API,而不用关注底层实现。

Servlet/JSP 动态编译

以下是追溯 createJspServlet 调用链:



可以看出,resin 在 PageFilterChain 做 doFilter 时,会判断当前请求的 jsp 或者 serlvet 对应的 class 是否存在,如果 java 文件存在而 class 不存在则进行动态编译. PageFilterChain 的主要作用之一也就是做动态代码生成, 生成的工作如上图,是交给了 ServletConfigImpl 来进行.Resin 为每个 Webapp 默认插入一个 PageFilterChain 过滤器,系统访问的每次请求就会首先经过 PageFilterChain 进行处理.

附录

运行时,远程调试

可通过运行时调试来捕捉 resin 行为这里需要打开 resin 调试端口并采用 jdb 挂接 resin JVM 进行调试.通过 jdb 调试,我们可以获取 resin 在运行时的包括线程调用堆栈等所有内容.

这里提供了一个脚本 debug_httpd.sh,放在 RESIN_HOME/bin 目录下:

```
#! /bin/sh
CUR_DIR=$(cd "$(dirname "$0")"; pwd);
$CUR_DIR/httpd.sh -Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=5432 &
$JAVA_HOME/bin/jdb -connect com.sun.jdi.SocketAttach:hostname=localhost,port=5432
```

通过 debug_httpd.sh 来启动 resin. 标本运行后,resin 先启动,接着就直接进入了 jdb 调试环境.

Resin 性能:

服务器缓存:

默认情况下,页面是没有被缓存的.如果要缓存页面,你必须在 HTTP 头中设置缓存标记值.
Resin 的缓存操作像一个代理缓存.缓存策略主要由 HTTP 头来决定.Resin 可以缓存 JSP, XTP 页面,主要取决于 response 返回的 headers 值.

Cache-Control: max-age

设置 max-age 头将会把结果缓存指定的时间.对于高负载页面,设置过期时间能够显著提高性能.
注意,使用 session 的页面不会被缓存,但是"Cache-Control: private"能够仅对 session 关联的浏览器进行缓存.
max-age 对于从数据库中产生的页面缓存较为适用。如果你是基于比如文件的修改时间来做缓存,那么就应该 If-Modified 头。

开启缓存调试

在 resin.conf 中加入如下配置项可开启缓存日志。

```
<log name="com.caucho.server.cache" path="log/cache.log" level="fine"/>
```

在缓存日至中可以看到大致如下的内容:

```
[10:18:11.369] caching: /images/caucho-white.jpg etag="AAAAPbkEyoA" length=6190
[10:18:11.377] caching: /images/logo.gif etag="AAAAOQ9zLeQ" length=571
[10:18:11.393] caching: /css/default.css etag="AAAANzMooDY" length=1665
...
[2003/09/12 10:18:49.303] using cache: /css/default.css
[2003/09/12 10:18:49.346] using cache: /images/pixel.gif
```

Expires

应用程序同样可以设置 Expires 头来开启缓存，用 Expires 指定的时间是绝对时间而非时间间隔。在采用 Expires,的页面中 Session 必须被禁用。Exipires 同样是对于数据库生成的页面较为适用.

If-Modified

可根据更改时间来缓存.只有在页面内容更改时缓存才会失效并更新.

Resin 中也可以对 Servlet 或 JSP 设置 If-Modified 来做缓存.

```
<%@ page session="false" %>
<%!
int counter;
public long getLastModified(HttpServletRequest req)
{
  String path = req.getRealPath("test.xml");
  return new File(path).lastModified();
}
%>
Count: <%= counter++ %>
```

Servlet 缓存

servlet 的缓存和 JSP 类似(或者 XTP 静态文件). Resin 的缓存机制工作模式类似代理缓存. 它不知道页面是如何生成的(静态或动态的);只要相应的缓存 headers 头设置了,目标页面就可以被缓存.

Session 和 Cookies

因为 Resin 遵循 Http 缓存规范,所以有 session 的页面同样也会被缓存.如下做一个简单测试:

```
<% response.addHeader("Cache-Control", "max-age=3600"); %>
session id <%= session.getId() %>
```

上面的缓存实际上是一个公共缓存页面, 如果这样,所有访问此页面的人都能看到你的 session 信息.为了避免此问题,你需要另外再设置一个"Cache-Control:private"头,让页面成为 session 私有的缓存页面.代码如下:

```
<%
  response.addHeader("Cache-Control", "max-age=3600");
  response.addHeader("Cache-Control", "private");
%>
session id <%= session.getId() %>
```

被 include 的页面

Resin 中,如果一个 include 子页面的父页面不能被缓存,子页面还是可以被缓存的,包括公共缓存和私有缓存.

Resin 把子页面当作独立的请求,所以缓存策略也是独立于父页面的.样例示范:

```
<% if (! session.isNew()) { %>
<h1>Welcome back!</h1>
<% } %>
<jsp:include page="expires.jsp"/>
```

Vary

很多情况下,对于同一个页面, 能识别 gzip 压缩文件的浏览器能够获取此页面的压缩版本,不识别压缩页面的浏览器只能获取其未压缩版本.通过使用 Vary 头,Resin 可以对同一个页面缓存不同的版本.

```
<%
   response.addHeader("Cache-Control", "max-age=3600");
   response.addHeader("Vary", "Accept-Encoding");
%>
Accept-Encoding: <%= request.getHeader("Accept-Encoding") %>
```

缓存匿名用户

登录用户会看到专门的定制页面,而匿名用户只能看到公共页面.在这种情况下,你需要做一些工作来优化缓存.
一种解决方案就是: include 公共子页面.父页面针对登录用户来定制,不设置缓存.

Experimental 匿名缓存

Resin 包含一个匿名用户缓存特性.如果一个用户未登录,他将获取一个缓存了的页面. 反之若登录了,他将会获取个人页面. 但是,如果使用了 cookies 来追踪匿名用户状态,此特性就不会有效工作.
如果要开启匿名缓存,需要设置 Cache-Control: x-anonymous 头.之后 Resin 将会对此页面用户做缓存.

```
<%@ page session="false" %>
<%! int counter; %>
<%
response.addHeader("Cache-Control", "max-age=15");
response.addHeader("Cache-Control", "x-anonymous");
String user = request.getParameter("user");
%>
User: <%= user %> <%= counter++ %>
```

顶级页面仍需设置 Expires 或 If-Modified 头,Resin 会决定此页面是否该被缓存. 假如请求中包含有 cookies 信息,Resin 就不会缓存此页面, 或者不会发送此页面的缓存版本.在未使用 cookie 的情况下 Resin 才会返回缓存页.

Cache-mapping

cache-mapping 会对可缓存的 If-Modified 页面赋予一个 max-age 和 Exipres 值,而不会影响可缓存的 max-age 或者 Expires 页面.
你可能需要对有些页面设置很长的 Expires 期限.比如,.gif 基本不会更改.但是在它一旦改变的同时,你又希望它能马上生效.
这里你可以对一个 gif 文件社会自 24 小时的 Expires 超时值, 这需要在默认的 FileServlet 中设置:

```
<web-app id='/'>
   <cache-mapping url-pattern='*.gif' expires='24h'/>
</web-app>
```

cache-mapping 自动生成 Expires 头. 它仅仅对设置了 If-Modified 或者 Etag 的页面生效.它不会对已经显式设置了 Expires 或者不可缓存的页面生效. 所以, 就算对 jsp 文件做此种设置也是很安全的.

高级日志配置

可以在 resin.conf 中添加如下配置节来记录 resin 运行时的所有 debug 日志,从而可以分析 resin 的运行状况, 比如进行 restart 的原因.

```
<log name="" level='finer' path="log/debug.log" timestamp="[%H:%M:%S.%s]" rollover-period="1D"/>
```

相应的,也可以在单个 webapp 的 WEB-INF/web.xml 中添加此类日志,以分析单个 webapp 的完整运行状况.如下,把完整的 debug 日写入<web-app-root>/WEB-INF/work/debug.log

```
<!-- <web-app-root>/WEB-INF/web.xml -->
<web-app>
  <log name='' level='finer' path='WEB-INF/work/debug.log'
       timestamp="[%H:%M:%S.%s]"
       rollover-period='1D'/>
  ...
</web-app>
```

用参数-Xloggc:gc.log 运行 Resin 可以在 RESIN_HOME 下生成 gc.log 日志

监控 HTTP 通信

在 resin.conf 中添加如下配置节:

```
<log name='com.caucho.server.http' level='finer' path='log/http.log'/>
<log name='com.caucho.server.connection' level='finer' path='log/http.log'/>
```

开启 Resin 调试端口:

```
$RESIN_HOME/bin/httpd -Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=5432
```

jdb 调试:

```
$JAVA_HOME/bin/jdb -connect com.sun.jdi.SocketAttach:hostname=localhost,port=5432
```

jdb will show something like:

Set uncaught java.lang.Throwable

Set deferred uncaught java.lang.Throwable

Initializing jdb ...

>

```
>suspend
>where  all
>resume
>quit
```

性能 FAQ

缓存

最重要的性能指标是在不经常变换内容的 servlet/JSP 输出中设置 Expires,最好是 Last-Modified 或者 ETag 值.

当一个 servlet forward 内容时什么被缓存了?

唯一的凭据就是 HTTP 头.所以如果你是 telnet 到 resin,你将看不到 header 是怎么设置的.

在 forward 过程中,你可以修改 header 而不用修改 JSP 本身.

有个需要注意的地方: 缓存是基于原始 URL 的.

可缓存意味着:
ETag 或者 Last-Modified 需要在响应头(ETag is better)中被设置了.

Resin 能够承载多少并发用户?
取决于应用程序类型,数据库的使用,session 对象的使用,服务器端缓存,应用程序架构.可以在 resin.conf 中调节 thread-max 的值.

Resin 采用了 NIO 吗?
Resin 没有采用 NIO,而是采用 JNI 来捕获底层 I/O 调用.性能会比 NIO 更好?

Resin 如何使用 JNI
JNI 代码针对各种平台而编译.Resin 在性能要求严格的地方使用 JNI,比如底层 socket 连接和文件操作. 同样,采用 JNI 来和 OpenSSL 库进行互操作.(其实只有 Pro 版本才能使用 JNI)
一个显著的好处就是 Resin 不需要多每个 keepalive 连接配一个单独的线程,这是通过采用底层的 poll()(或者 select())函数来做到的.最终结果可能是多个连接对应单个线程.

如果 JNI 不可用那么就会使用等价的 JDK 调用. 同样, OpenSSL 仅只能通过 JNI 方式启用.
Resin 会在启动时打印一条日志来指示 JNI 是否启用:
Loaded Socket JNI library.
If JNI is not available, the log message is:
Socket JNI library is not available