

GWT 开发者手册

格式规范：这种字体表示代码或术语

文档目的：对新进员工进行 GWT 技术培训，使他们可以在以前没有接触过 GWT 的基础上，通过本文档，经过短期学习，即可使用 GWT 进行日常开发工作。

GWT 版本：gwt-windows-1.3.3

面向读者：熟悉 JAVA 语言，有用 J2EE 开发三层架构软件系统的经验 WEB 应用程序开发者。

一、基本概念

核心 GWT 概念，诸如：把 JAVA 原码编译为 JAVASCRIPT 原码，调试，跨浏览器支持，和定义模块（module）等。

GWT 编译器

主要用于把 JAVA 应用程序转换成对应的 JAVASCRIPT 应用程序

GWT 的核心是一个编译器，它可以把 JAVA 代码转换成 JAVASCRIPT 代码，把用于实际开发工作的 JAVA 应用程序转变成等价的 JAVASCRIPT 应用程序，一般来说：

1. 如果你的 GWT 应用在主机模式中编译运行，
2. 并且 GWT 把你的应用程序编译成 JAVASCRIPT 应用程序，而没有报错，
3. 那么你的应用程序将会以同样的方式在 WEB 浏览器中工作，就像在主机模式中一样。

GWT 编译器支持大多数 JAVA 语言特性。GWT 运行时库仿真了 JAVA 运行时库的一个子集。

注意：

1. 语言支持：

GWT 可以编译 J2SE 1.4.2 或更早版本。但有一些不一致需要注意。

- 固有类型

`byte`, `char`, `short`, `int`, `long`, `float`, `double`, `Object`, `String`, 和数组都是支持的。毕竟，在 JAVASCRIPT 语言中没有 64 位整型，所以 `long` 类型变量被映射到 JAVASCRIPT 的双精度浮点类型值。为了保证主机模式和 WEB 模式中最大的一致性，我们推荐使用 `int` 类型变量

- 异常

`try`, `catch`, `finally` 和用户自定义的异常都可以正常支持，但是注意，在 WEB 模式中 `Throwable.printStackTrace()` 方法不被支持。

- 断言

GWT 编译器解析 JAVA `assert` 声明，但是它们不会被生成对应的 JAVASCRIPT 代码。

- 多线程和同步

JAVASCRIPT 解析器是单线程的，所以 GWT 自然接受 `synchronized` 关键字，但是没有实际作用。同步相关的类库方法也不可用，包括 `Object.wait()`, `Object.notify()`, 和 `Object.notifyAll()`

- 反射

为了效率最大化，GWT 把你的 JAVA 原代码编译成单块集成脚本，并且不支持类的并发动态装载（我也不大懂是怎么回事，总之不要在 GWT 中使用 JAVA 的反射机制），另外，其他最佳化选项也不包括对反射的一般支持。但是可以使用 `GWT.getTypeName(Object)` 来查询一个对象的类名。

- 无用对象清理

JAVASCRIPT 不支持垃圾回收阶段的对象清理，所以 GWT 不能支持 WEB 模式下的 JAVA 对象清理。

- 严格浮点

JAVA 语言规范精确地定义了浮点支持，包括单精度和双精度，还有 `strictfp` 关键字。GWT 不支持 `strictfp` 关键字，并且不能保证在转换后的代码中的浮点的详细精度，所以如果你需要保证精度的计算，则要避免在客户端代码中进行。

2. 运行时库支持

GWT 只支持 JAVA2 标准版和企业版类库的一个小的子集，由于这些库非常大，并且有很多在 WEB 浏览器中不可用的功能。详细请参考：`java.lang` and `java.util` 的 API，那里列出了被支持的类和两个类库（GWT 和标准 JAVA 类库）行为上的不同。

一些特定领域 GWT 与 JAVA 标准运行时不同。

- 正则表达式

JAVA 正则表达式的语法类似但也不同于 JAVASCRIPT 的正则表达式语法。例如，`replaceAll` 和 `split` 方法。你可能只想使用和 JAVASCRIPT 同效的 JAVA 正则表达式。

- 序列化

JAVA 的序列化依赖于很少的机制，但这些机制在编译后的 JAVASCRIPT 中不可用，例如动态类装载和反射。所以 GWT 不支持标准 JAVA 序列化，但是 GWT 有一个 RPC 机制，它提供了调用远程方法时与服务器交互的自动对象序列化。

注意：如果你确保在一开始你的客户端代码中只使用了可以转换的类，你将会避免很多麻烦。为了能够尽早确定问题所在，你的代码将会在主机模式中运行时被检查是否符合 JRE 仿真库。所以，不支持的类库将会在第一次运行应用程序时被检测到。应该尽早运行并经常运行你的程序。

跨浏览器支持

GWT 架构可以在单一代码的基础上支持多浏览器。

GWT 使你不用过于担心浏览器的不相容性。如果你关注嵌入式的界面元素和组件，你的应用程序在最近版本的 Internet Explorer, Firefox, 和 Safari 浏览器中的表现是一致的（大多数情况下，Opera 浏览器也是这样）。DHTML 用户界面是非常复杂的，虽然这样，也要确定在每个浏览器中彻底测试你的应用程序。

只要可能，GWT 符合浏览器的本地用户界面元素。例如：GWT 的 `Button` 元素是一个真正的 HTML `<button>`，而不是一个合成的类似按钮的界面元素，比如，一个 `<div>`。这说明 GWT 按钮在不同的浏览器和不同的客户端操作系统中都能恰当地显示。我们喜欢本地浏览器控制是因为它们是快速、可用并且被用户熟悉。

当开发一个风格化的 WEB 应用程序，CSS 是非常理想的。开发者应该在样式表中定义样式，这些样式使用样式名（`style names`）链接到应用程序代码。

在主机模式中调试

GWT 有一个嵌入式的 DHTML 浏览器，你可以在转换成 JAVASCRIPT 应用之前，在任何 JAVA 开发环境中运行和调试你的应用程序。

你将会花费你的大部分的开发时间在主机模式上，这是说你和你的 GWT 应用程序互动时，不需要先把它转换成 JAVASCRIPT。任何时候你在 JAVA 集成开发环境中编辑、运行和调试应用程序，你都是在主机模式下工作。当运行在主机模式，JAVA 虚拟机实际上是在执行编译后的 JAVA 字节码，使用 GWT 输出到一个嵌入的浏览器窗口。在这种传统的“编码—测试—调试”的开发周期中，主机模式是快速开发你的应用的最有效的方式。

启动一个主机模式会话，你的启动类应该是

`com.google.gwt.dev.GWTShell`，这个类可以在 `gwt-dev-windows.jar` (或 `gwt-dev-linux.jar`) 中找到。

小提示：

在主机模式下，GWT 开发命令工具（development shell）会使用虚拟机的类路径查找模块（和客户端源代码）。在运行 GWT 开发命令工具（development shell）时要确保把源代码路径加入到了你的类路径中。

在 WEB 模式中部署

为便于部署，首先把你的应用编译成 JAVASCRIPT。

当你从开发阶段转移到端到端测试和生产，你会开始更多地和你的应用程序在 WEB 模式下交互。WEB 模式是指从通常的浏览器来访问你的应用程序—在浏览器中它作为纯净的 JavaScript 运行—正如它最终要被部署的样子。

为你的模块创建一个 WEB 模式版本，你需要使用主机模式浏览器中的“Compile/Browse”按钮或命令行编译器

```
com.google.gwt.dev.GWTCompiler.
```

WEB 模式论证了是什么使 GWT 不同凡响：当你的应用在 WEB 模式中启动时，它完全作为 JAVASCRIPT 运行，并且不需要任何浏览器插件或 JVM。

HTML 主页

一个 HTML 文档形式的主页，它包含 GWT 模块。

任何符合适当规范的 HTML 页面可以包含 GWT 创建的代码，作为一个主页被引用。一个经典的 HTML 主页如下所示：

```
<html>
  <head>
    <!-- The fully-qualified module name -->
    <meta name='gwt:module'
    content='com.example.cal.Calendar'>
    <!-- Properties can be specified to influence deferred binding
    -->

    <meta name='gwt:property' content='locale=en_UK'>
    <!-- Stylesheets are optional, but useful -->
    <link rel="stylesheet" href="Calendar.css">
    <!-- Titles are optional, but useful -->
    <title>Calendar App</title>
  </head>
  <body>
```

```
<!-- Include the bootstrap script just inside the body or in
the head -->
<!-- (startup is slightly faster if you place it just after
the body tag -->
<script language="javascript" src="gwt.js"></script>
<!-- Include a history iframe to enable full GWT history
support -->
<!-- (the id must be exactly as shown) -->
<iframe id="__gwt_historyFrame"
style="width:0;height:0;border:0"></iframe>
</body>
</html>
```

这个结构是可以向现存的 **WEB** 应用程序添加功能而把改动最小化。

客户端代码

“客户端”意思是指将要被转换并在 **WEB** 浏览器中以 **JAVASCRIPT** 形式运行的原代码。

你的应用程序将要被通过网络送向客户，在那里它作为 **JAVASCRIPT** 运行在 **WEB** 浏览器中。用户浏览器中所发生的一切，可以看作是客户端处理。当你写在 **WEB** 浏览器中运行的客户端代码时，记住它们最终要变成 **JAVASCRIPT**。所以，要使用那些可以被转换的类库和 **JAVA** 语言结构是非常重要的。

服务端代码

“服务端”意思是指不会被转换，并且只作为字节码运行在服务器端的原代码。

发生在服务器端的一切可以被看作是服务器端处理。当你的应用程序需要与服务器互动（例如，上载或下载数据），这会产生一个穿过网络的客户端请求（从浏览器）使用 **remote procedure call (RPC)**。在进行处理时 **RPC**，服务器要执行服务端代码。

小提示：

GWT 不会去管你的服务器运行 **JAVA** 字节码的能力。服务端代码不需要被转换，所以你可以使用你认为有用的任何 **JAVA** 类库。

项目结构

GWT 项目由一个推荐包布局构建而成。

GWT 项目以 **JAVA** 包的方式进行布局，这样，大多数的配置可以从类路径（`classpath`）和你的模块定义（`module definitions`）中推导出。

如果你要从代码片断开始一个 **GWT** 项目，你应该使用标准 **GWT** 包结构，这种结构可以很容易地区分客户端代码和服务端代码。例如：假定你的新项目叫“**Calendar**”。则标准包结构会如下所示：

包	目的
<code>com/example/cal/</code>	项目根包，包含模块 XML 文件
<code>com/example/cal/client/</code>	客户端代码文件和子包
<code>com/example/cal/server/</code>	服务端代码和子包
<code>com/example/cal/public/</code>	静态资源

例子文件组织如下：

文件	目的
<code>com/example/cal/Calendar.gwt.xml</code>	一个通用基本模块，用于你的项目，它继承了 <code>com.google.gwt.user.User</code> 模块
<code>com/example/cal/CalendarApp.gwt.xml</code>	继承了 <code>com.example.cal.Calendar</code> 模块 (见上)并且加入一个入口类
<code>com/example/cal/CalendarTest.gwt.xml</code>	一个你的项目定义的模块
<code>com/example/cal/client/CalendarApp.java</code>	客户端 JAVA 原代码,用于入口类。
<code>com/example/cal/client/spelling/SpellingService.java</code>	一个定义在子包中的 RPC 服务

接口

<code>com/example/cal/server/spelling/SpellingServiceImpl.java</code>	服务端 JAVA 原代码,它实现了后台服务业务逻辑。
<code>com/example/cal/public/Calendar.html</code>	一个 HTML 页面,用于装载应用程序。
<code>com/example/cal/public/Calendar.css</code>	一个样式表,用于风格化应用程序。
<code>com/example/cal/public/images/logo.gif</code>	一个题头标志

模块

模块是一个 **XML** 文件,它包含与应用程序或类库相关的设置。

一个 GWT 配置的单独单元,是 **XML** 格式的文件。包括所有的你的 GWT 项目需要的配置信息,即下列信息:

- 继承模块
- 一个入口点应用类名;这些是可选的,虽然任何关系到 **HTML** 的模块都必须至少有一个指定的入口类。
- 原代码路径
- 公共路径
- 延期绑定规则,包括属性提供者和类生成器。

模块 (**Modules**) 可以出现在你的类路径的任何包里,但是强力推荐它应该出现在标准项目布局的根包里。

入口类

一个模块入口是任何实现 `EntryPoint` 接口的类,并且可以被无参数构造实例。当装载一个模块时,每个入口类被实例化,并且它们的 `EntryPoint.onModuleLoad()` 方法被调用。

原代码路径

模块能够指定哪个子包包含可转换原代码,方法是把命名包和它的子包被加入原代码路径。只有在建立在原代码路径上的文件才可能被转换成 **JAVASCRIPT**,客户端代码和服务端代码也可以无冲突地混合在同一个类路径中。

当模块继承其他模块，它们的源代码路径被绑定，这样每个模块将会能够访问到它需要的可转换源代码。

公共路径

模块能够指定哪个子包是公共的，方法是命名包和它的子包被加入到公共路径。当你把你的应用程序编译成 **JAVASCRIPT** 时，在公共路径中能找到的所有的文件都被复制到模块的输出目录。净效果是用户可见的 **URLs** 不需要包含一个完整的包名。

当模块继承其他模块时，它们的公共路径被绑定，这样每个模块都可以访问它所需要的静态资源。

特殊规范

- 模块 XML 格式

在 **XML** 文件中定义模块，并且置入你的项目包层级。

模块定义在 **XML** 文件中，它的文件名扩展是 `.gwt.xml`。

模块 **XML** 文件应当位于你的项目的根包

如果你正在使用标准项目结构，你的模块 **XML** 可以像以下这样简单：

```
<module>
<inherits name="com.google.gwt.user.User"/>
<entry-point class="com.example.cal.client.CalendarApp"/>
</module>
```

- 装载模块

在 **JAVA** 类路径中发现的模块 **XML** 文件，被它们的逻辑模块引用，从主页用名字引入，也被其他模块引入。

模块总是关联到它们的逻辑名。模块的逻辑名遵守这种形式 `pkg1.pkg2.ModuleName` 且不包括实际文件系统路径和文件扩展名。例如：模块 **XML** 文件的逻辑名位于：

```
~/src/com/example/cal/Calendar.gwt.xml
```

是

`com.example.cal.Calendar`

● 可用元素

`<inherits name="logical-module-name"/>`

从指定的模块继承所有的设置，就像被继承的模块的 XML 被逐字复制。一些模块可被用这种方式继承。

`<entry-point class="classname"/>`

指定的入口点类。一些入口类能够被从被继承模块加入、包含。 `<source path="path"/>`

通过绑定包向原代码路径加入包，在包中，模块 XML 可以在特定的子包路径中找到。任何出现在这个子包下的 JAVA 原代码文件，或任何下层子包内，假定都是需要被转换的。

如果在模块 XML 文件中没有定义 `<source>` 这个元素，客户 (client) 子包会被隐含地原代码路径，就像模块 XML 文件中有定义 `<source path="client">` 一样。这个默认设置帮助保持模块 XML 使用标准项目结构。

`<public path="path"/>`

通过绑定包向公共路径中加入包，在绑定包中，模块 XML 将会在指定路径中被发现来指定

`<servlet path="url-path" class="classname"/>`

为了 RPC 测试的方便性，这个元素装载一个 SERVLET 类，它应用到指定的 URL 路径。这个 URL 路径应该是绝对路径，并且符合目录形式（例如： `/spellcheck`）。你的客户端代码指定这个 URL 映射到一个调用 `ServiceDefTarget.setServiceEntryPoint(String)`。一些 SERVLET 可以用这种方式装载，包括那些从继承模块里来的。

`<script src="js-url">script ready-function body</script>`

自动注入外部的 JAVASCRIPT 文件，这个文件位于 `src` 指定的位置。`script ready-function body` 是一个 JAVASCRIPT 函数体，当这个脚本已知被初始化时，它返回 `true`

`<stylesheet src="css-url"/>`

自动注入外部的 CSS 文件， 这个文件位于 `src` 指定的位置。

`<extend-property name="client-property-name" values="comma-separated-values"/>`

为一个已存在的客户属性值集合进行扩展。一些值可以用这种方式添加，并且客户属性值通过继承模块进行累积。你可能只是发现它对于“在国际化中指定本地”([specifying locales in internationalization](#))有用

- 自动资源注入

模块组 (**Modules**) 能够包含到外部 JAVASCRIPT 和 CSS 的引用，方法是当模块自身装载时，它们也被自动装载。

模块可以包含到外部 JAVASCRIPT 和 CSS 文件的引用，方法是当模块自我加载时自动加载。

- 注入外部 JAVASCRIPT

对于在你的模块自动地关联外部 JAVASCRIPT 文件，脚本注入是一个方便的方法。脚本注入使用下面的语法：

```
<script src="js-url"><![CDATA[
    script ready-function body
]]></script>
```

“*ready-function body*” 部分是 JAVASCRIPT 函数主体部分，当脚本被装载并可用后，它返回 `true`。脚本被装载进主页 ([host page](#)) 的名字空间 ([namespace](#))，其作用与你显式地使用 HTML `<script>` 元素进行引入是相同的。

例如：假定你的模块需要的脚本名为 `:InjectedScript.js`。那么示例代码如下：

```
function foo() {
    // do something neat
    doSomethingTimeConsuming();
}
function bar() {
    // do something else neat
}
```

你的模块应该如下所示：

```
<module>
    <inherits name="com.google.gwt.user.User"/>
    <script src="InjectedScript.js"><![CDATA[
// More complex tests are possible, but usually checking for
the existence
```

```

    // of a function is enough.
    if ($wnd.bar)
        return true;
    else
return false;
    ]]></script>
</module>

```

可用一函数（**ready-function**）的目的就是明确地指出脚本已被完全装载，这样，你的 GWT 代码就能够使用 [JSNI](#)，并且可以确定被引用的标识符是可用的。在上面的例子中，函数 bar 的存在就说明了脚本已经就绪了。

- 注入外部的样式表

样式表注入是把外部的 **CSS** 文件自动关联到你的模块的一个便捷方式。使用如下语法可以把 **CSS** 文件自动附加到你的主页（[host page](#)）上。

```
<stylesheet src="css-url"/>
```

你能够用这种方式加入一些样式表，包含入页面时的顺序就是元素在你的模块 **XML** 文件中出现的顺序。

- 注入和模块继承

模块继承使资源注入更加方便。如果你想要创建一个可重用类库，这个库依赖特定的样式表的 **JAVASCRIPT** 文件，你可以确定你的类库的客户可以用从模块继承的方式自动得到所需之物。

- 过滤公共包

在你的公共路径中滤进或滤出文件，以避免无意地发布文件。

`<public>` 元素不支持完全的 `FileSet` 语义。当前只有下列属性和嵌套元素被支持：

- `includes` 属性
- `excludes` 属性
- `defaultexcludes` 属性
- `casesensitive` 属性

- 嵌套标志 `include`

- 嵌套标志 `exclude`

其他属性和嵌套元素不被支持

🚦 重要注意事项

`Defaultexcludes` 的默认值是 `true`。

命令行工具

开始开发时需要的一些有用的命令行工具。

GWT 只有很少的几个命令行工具。它们也可以用于向现存项目加入新的东西。

例如，`projectCreator` 可以用于使一个 **Eclipse** 项目符合 **GWT** 的规范。

● `projectCreator`

生成基本的项目骨架。一个可选的 **Ant build** 文件，和 / 或 **Eclipse** 项目。

```
projectCreator [-ant projectName] [-eclipse projectName]  
[-out dir] [-overwrite] [-ignore]
```

<code>-ant</code>	生成一个 Ant build 文件，用于编译源代码（将会加入 <code>.ant.xml</code> ）
-------------------	---

<code>-eclipse</code>	生成一个 eclipse 项目。
-----------------------	-------------------------

<code>-out</code>	输出文件写入到这个目录（默认是当前目录）
-------------------	----------------------

<code>-overwrite</code>	覆盖任何已经存在的文件。
-------------------------	--------------

`-ignore` 忽略任何已经存在的文件；不覆盖

■ 示例

```
~/Foo> projectCreator -ant Foo -eclipse Foo
Created directory src
Created directory test
Created file Foo.ant.xml
Created file .project
Created file .classpath
```

运行 `ant -f Foo.ant.xml` 将会把 `src` 编译到 `bin`。这个 `build` 文件也包含一个目的包，这个包用于把项目打包成一个 `jar` 文件。

`.project` 能够被引入到 **Eclipse** 工作区

● applicationCreator

生成一个初始的应用程序，这个应用程序可以从主机模式启动，并且可以编译成 `JAVASCRIPT`

```
applicationCreator [-eclipse projectName] [-out dir]
[-overwrite] [-ignore] className
```

`-eclipse` 为命名的 **eclipse** 项目生成一个调试启动配置项。

`-out` 这个目录用于写入输入文件（默认为当前目录）

`-overwrite` 覆盖任何已经存在的文件

`-ignore` 忽略任何已经存在的文件；不覆盖

`className` 创建一个合乎规格应用程序类名字

■ 示例

```
~/Foo> applicationCreator -eclipse Foo
com.example.foo.client.Foo
Created directory src/com/example/foo/client
Created directory src/com/example/foo/public
Created file src/com/example/foo/Foo.gwt.xml
Created file
src/com/example/foo/public/Foo.html
Created file
src/com/example/foo/client/Foo.java
Created file Foo.launch
Created file Foo-shell
Created file Foo-compile
```

运行 `Foo-shell` 命令行在主机模式中生成新的应用。`Foo-compile` 把 JAVA 应用转换成 `JAVASCRIPT`，在目录 `www.Foo.launch` 下创建一个 WEB 文件夹，这是一个用于 `Eclipse` 的启动配置。

● junitCreator

生成 JUnit 测试程序

生成一个 JUnit `test` 测试程序和脚本，可用于主机模式和 **WEB** 模式。

```
junitCreator -junit pathToJUnitJar [-eclipse projectName]
[-out dir] [-overwrite] [-ignore] className
```

- | | |
|-----------------------|--|
| <code>-junit</code> | 指定到你的 <code>junit.jar</code> 的路径（必须） |
| <code>-module</code> | 指定要使用的应用模块名（必须）。 |
| <code>-eclipse</code> | 为命名的 <code>eclipse</code> 项目创建一个调试启动选项 |
| <code>-out</code> | 用于写入输出文件的目录（默认是当前目录） |

<code>-overwrite</code>	覆盖任何已经存在的文件。
<code>-ignore</code>	忽略任何已经存在的文件；不覆盖
<code>-createMessages</code>	为消息（ Messages ）接口生成脚本，而不是一个常量。
<code>interfaceName</code>	要创建的完全符合规范的接口名字。

■ 示例

```
~/Foo> il8nCreator -eclipse Foo
-createMessages
com.example.foo.client.FooMessages
Created file
src/com/example/foo/client/FooMessages.properties
Created file FooMessages-il8n.launch
Created file FooMessages-il8n

~/Foo> il8nCreator -eclipse Foo
com.example.foo.client.FooConstants
Created file
src/com/example/foo/client/FooConstants.properties
Created file FooConstants-il8n.launch
Created file FooConstants-il8n
```

运行 `FooMessages-il8n` 将会从 `FooMessages.properties` 产生一个接口，

这个接口扩展了 `Messages`（消息会需要参数，用第 `N` 个参数，替换 `{n}`）运行

`FooConstants-il8n` 将会从 `FooConstants.properties` 生成一个接口
（常量不会使用参数）

在 Eclipse 项目中，这个启动配置和脚本有同样的效果。

二、构建用户界面

正如在画廊（`the gallery`）中展示的，GWT 包含了各种预先构建的 JAVA 界面元素（`widgets`）和面板（`panels`），它们可以为你的应用程序提供跨浏览器的构建块。

GWT 用户界面类是类似于已有的用户界面框架的，例如 `Swing` 和 `SWT`。除了界面元素（`widgets`）是用动态创建 HTML 的方式进行显示而不是直接面向像素的图形。

当需要操作浏览器的 DOM 可以直接使用 DOM 接口，更方便的方式是使用界面元素的类层级结构。你应该很少需要直接访问 DOM。使用界面元素会是更方便快捷构建界面，而且在所有的浏览器中都能正确工作。

- 特别规范

- 界面元素和面板

界面元素和面板是客户端 JAVA 类，用于构建用户界面。

GWT 应用程序使用界面元素（`widgets`）构建用户界面，界面元素包含在面板（`panels`）内。界面元素的例子有：`Button`，`TextBox`，和 `Tree` 等。

界面元素和面板在所有的浏览器中都有同样的表现。使用它们，你可以避免对每种浏览器写特定的代码。但是你会受到工具箱所提供的界面元素集合的限制。有若干种方式使你可以创建自定义的界面元素。

- 面板

面板，诸如 `DockPanel`，`HorizontalPanel`，和 `RootPanel`，包含界面元素，并且用于定义如何在浏览器中对界面元素进行布局。

- 样式

视觉风格通过级联样式表（CSS）应用到界面元素上。具体做法如下：

每个界面元素都有一个关联的样式名，它绑定到对应的 CSS 规则。一个界面元素的样式名使用 `setStyleName()` 来设置。例如：`Button` 元素的默认样式名是 `gwt-Button`。

为了给所有的按钮一个更大的字体，你可以把下列规则加入到你的应用程序 CSS 文件中。

```
.gwt-Button { font-size: 150%; }
```

🚦 复杂样式

一些界面元素有稍微复杂的样式。例如：MenuBar，有下列样式：

```
.gwt-MenuBar { the menu bar itself }
.gwt-MenuBar .gwt-MenuItem { menu items }
.gwt-MenuBar .gwt-MenuItem-selected { selected menu items }
```

在这个例子中，有两个样式规则应用到菜单项目。第一个应用到所有的菜单项（已选中或未选中的），同时，第二个（有**-selected** 后缀）应用到已经选中的菜单项。一个选中的菜单项的样式名字会设置成"`gwt-MenuItem`

`gwt-MenuItem-selected`"，指定这两种被应用的样式规则。最通常的方式

是使用 `setStyleName` 来设置基本样式名，然后用 `addStyleName()` 和

`removeStyleName()` 来添加和去除第二个样式名。

🚦 CSS 文件

通常，样式表被置于你的模块的公共路径下的包里。然后简单地你的主页中包含它，如下：

```
<link rel="stylesheet" href="mystyles.css" type="text/css">
```

■ 界面元素画廊

一个界面元素和面板的展示区

下列界面元素和面板都可以 GWT 用户界面库中找到。

🚦 按钮 (Button)



示例代码：

```
public class ButtonExample implements EntryPoint {
    public void onModuleLoad() {
        // Make a new button that does something when you click it.
        Button b = new Button("Jump!", new ClickListener() {
            public void onClick(Widget sender) {
                Window.alert("How high?");
            }
        });
    }
}
```

```

    }
    });
    // Add it to the root panel.
    RootPanel.get().add(b);
  }
}

```

✚ 单选按钮 (RadioButton)

☒ Choice 1 ☐ Choice 2 (Disabled)

示例代码:

```

public class RadioButtonExample implements EntryPoint {
    public void onModuleLoad() {
        // Make some radio buttons, all in one group.
        RadioButton rb0 = new RadioButton("myRadioGroup", "foo");
        RadioButton rb1 = new RadioButton("myRadioGroup", "bar");
        RadioButton rb2 = new RadioButton("myRadioGroup", "baz");
        // Check 'baz' by default.
        rb2.setChecked(true);
        // Add them to the root panel.
        FlowPanel panel = new FlowPanel();
        panel.add(rb0);
        panel.add(rb1);
        panel.add(rb2);
        RootPanel.get().add(panel);
    }
}

```

✚ 复选框 (CheckBox)

☐ Normal Check ☐ Disabled Check

示例代码:

```

public class CheckBoxExample implements EntryPoint {
    public void onModuleLoad() {
        // Make a new check box, and select it by default.
        CheckBox cb = new CheckBox("Foo");
        cb.setChecked(true);
        // Hook up a listener to find out when it's clicked.
        cb.addClickListener(new ClickListener() {
            public void onClick(Widget sender) {
                boolean checked = ((CheckBox) sender).isChecked();
                Window.alert("It is " + (checked ? "" : "not") +
                    "checked");
            }
        });
    }
}

```

```

    });
    // Add it to the root panel.
    RootPanel.get().add(cb);
}
}
}

```

文本框 (TextBox)



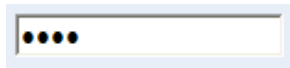
代码示例:

```

public class TextBoxExample implements EntryPoint {
    public void onModuleLoad() {
        // Make some text boxes. The password text box is identical
        // to the text
        // box, except that the input is visually masked by the
        // browser.
        PasswordTextBox ptb = new PasswordTextBox();
        TextBox tb = new TextBox();
        // Let's disallow non-numeric entry in the normal text box.
        tb.addKeyboardListener(new KeyboardListenerAdapter() {
            public void onKeyPress(Widget sender, char keyCode, int
            modifiers) {
                if (!Character.isDigit(keyCode)) {
                    // TextBox.cancelKey() suppresses the current
                    // keyboard event.
                    ((TextBox)sender).cancelKey();
                }
            }
        });
        // Let's make an 80x50 text area to go along with the other
        // two.
        TextArea ta = new TextArea();
        ta.setCharacterWidth(80);
        ta.setVisibleLines(50);
        // Add them to the root panel.
        VerticalPanel panel = new VerticalPanel();
        panel.add(tb);
        panel.add(ptb);
        panel.add(ta);
        RootPanel.get().add(panel);
    }
}

```

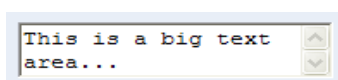
🚦 密码框 (PasswordTextBox)



示例代码:

```
public class TextBoxExample implements EntryPoint {
    public void onModuleLoad() {
        // Make some text boxes. The password text box is identical
        // to the text
        // box, except that the input is visually masked by the
        browser.
        PasswordTextBox ptb = new PasswordTextBox();
        TextBox tb = new TextBox();
        // Let's disallow non-numeric entry in the normal text box.
        tb.addKeyboardListener(new KeyboardListenerAdapter() {
            public void onKeyPress(Widget sender, char keyCode, int
            modifiers) {
                if (!Character.isDigit(keyCode)) {
                    // TextBox.cancelKey() suppresses the current
                    keyboard event.
                    ((TextBox) sender).cancelKey();
                }
            }
        });
        // Let's make an 80x50 text area to go along with the other
        two.
        TextArea ta = new TextArea();
        ta.setCharacterWidth(80);
        ta.setVisibleLines(50);
        // Add them to the root panel.
        VerticalPanel panel = new VerticalPanel();
        panel.add(tb);
        panel.add(ptb);
        panel.add(ta);
        RootPanel.get().add(panel);
    }
}
```

🚦 文本区 (TextArea)



示例代码:

```
public class TextBoxExample implements EntryPoint {
```

```

public void onModuleLoad() {
    // Make some text boxes. The password text box is identical to
the text
    // box, except that the input is visually masked by the browser.
    PasswordTextBox ptb = new PasswordTextBox();
    TextBox tb = new TextBox();
    // Let's disallow non-numeric entry in the normal text box.
    tb.addKeyListener(new KeyListenerAdapter() {
        public void onKeyPress(Widget sender, char keyCode, int
modifiers) {
            if (!Character.isDigit(keyCode)) {
                //
                TextBox.cancelKey() suppresses the current keyboard event.
                ((TextBox)sender).cancelKey();
            }
        }
    });
    // Let's make an 80x50 text area to go along with the other two.
    TextArea ta = new TextArea();
    ta.setCharacterWidth(80);
    ta.setVisibleLines(50);
    // Add them to the root panel.
    VerticalPanel panel = new VerticalPanel();
    panel.add(tb);
    panel.add(ptb);
    panel.add(ta);
    RootPanel.get().add(panel);
}
}

```

超级链接 (Hyperlink)

[Info](#)

[Buttons](#)

[Menus](#)

[Images](#)

[Layouts](#)

示例代码:

```

public class HistoryExample implements EntryPoint,
HistoryListener {
    private Label lbl = new Label();

```

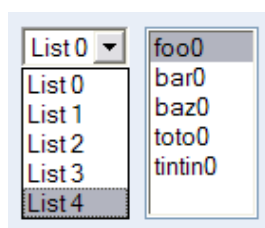


```

    public void onModuleLoad() {
// Create three hyperlinks that change the application's
history.
        Hyperlink link0 = new Hyperlink("link to foo", "foo");
        Hyperlink link1 = new Hyperlink("link to bar", "bar");
        Hyperlink link2 = new Hyperlink("link to baz", "baz");
        // If the application starts with no history token, start
it off in the
        // 'baz' state.
        String initToken = History.getToken();
        if (initToken.length() == 0)
            initToken = "baz";
        // onHistoryChanged() is not called when the application
first runs. Call
        // it now in order to reflect the initial state.
        onHistoryChanged(initToken);
        // Add widgets to the root panel.
        VerticalPanel panel = new VerticalPanel();
        panel.add(lbl);
        panel.add(link0);
        panel.add(link1);
        panel.add(link2);
        RootPanel.get().add(panel);
        // Add history listener
        History.addHistoryListener(this);
    }
    public void onHistoryChanged(String historyToken) {
        // This method is called whenever the application's history
changes. Set
        // the label to reflect the current history token.
        lbl.setText("The current history token is: " +
historyToken);
    }
}

```

🚦 列表框 (ListBox)



示例代码:

```

public class ListBoxExample implements EntryPoint {
    public void onModuleLoad() {
        // Make a new list box, adding a few items to it.
        ListBox lb = new ListBox();
        lb.addItem("foo");
        lb.addItem("bar");
        lb.addItem("baz");
        lb.addItem("toto");
        lb.addItem("tintin");
        // Make enough room for all five items (setting this value to
1 turns it
        // into a drop-down list).
        lb.setVisibleItemCount(5);
        // Add it to the root panel.
        RootPanel.get().add(lb);
    }
}

```

菜单条 (MenuBar)



示例代码:

```

public class MenuBarExample implements EntryPoint {
    public void onModuleLoad() {
        // Make a command that we will execute from all leaves.
        Command cmd = new Command() {
            public void execute() {
                Window.alert("You selected a menu item!");
            }
        };
        // Make some sub-menus that we will cascade from the top menu.
        MenuBar fooMenu = new MenuBar(true);
        fooMenu.addItem("the", cmd);
        fooMenu.addItem("foo", cmd);
        fooMenu.addItem("menu", cmd);
        MenuBar barMenu = new MenuBar(true);
        barMenu.addItem("the", cmd);
        barMenu.addItem("bar", cmd);
    }
}

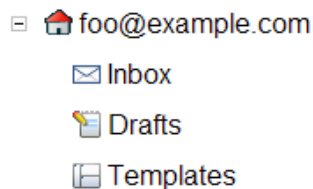
```

```

        barMenu.addItem("menu", cmd);
        MenuBar bazMenu = new MenuBar(true);
        bazMenu.addItem("the", cmd);
        bazMenu.addItem("baz", cmd);
        bazMenu.addItem("menu", cmd);
        // Make a new menu bar, adding a few cascading menus to it.
        MenuBar menu = new MenuBar();
        menu.addItem("foo", fooMenu);
        menu.addItem("bar", barMenu);
        menu.addItem("baz", bazMenu);
        // Add it to the root panel.
        RootPanel.get().add(menu);
    }
}

```

树 (Tree)



示例代码:

```

public class TreeExample implements EntryPoint {
    public void onModuleLoad() {
        // Create a tree with a few items in it.
        TreeItem root = new TreeItem("root");
        root.addItem("item0");
        root.addItem("item1");
        root.addItem("item2");
        // Add a CheckBox to the tree
        TreeItem item = new TreeItem(new CheckBox("item3"));
        root.addItem(item);
        Tree t = new Tree();
        t.addItem(root);
        // Add it to the root panel.
        RootPanel.get().add(t);
    }
}

```

表格 (Table)

sender	email
markboland05	mark@example.com
Hollie Voss	hollie@example.com
boticario	boticario@example.com
Emerson Milton	emerson@example.com
Healy Colette	healy@example.com
Brigitte Cobb	brigitte@example.com
Elba Lockhart	elba@example.com

🚦 表格页切换条 (TabBar)



示例代码:

```
public class TabBarExample implements EntryPoint {
    public void onModuleLoad() {
        // Create a tab bar with three items.
        TabBar bar = new TabBar();
        bar.addTab("foo");
        bar.addTab("bar");
        bar.addTab("baz");
        // Hook up a tab listener to do something when the user
        selects a tab.
        bar.addTabListener(new TabListener() {
            public void onTabSelected(SourcesTabEvents sender, int
            tabIndex) {
                // Let the user know what they just did.
                Window.alert("You clicked tab " + tabIndex);
            }
            public boolean onBeforeTabSelected(SourcesTabEvents
            sender,
                int tabIndex) {
                // Just for fun, let's disallow selection of 'bar'.
                if (tabIndex == 1)
                    return false;
                return true;
            }
        });
        // Add it to the root panel.
```

```

        RootPanel.get().add(bar);
    }
}

```

对话框 (DialogBox)



示例代码:

```

public class DialogBoxExample implements EntryPoint, ClickListener
{
    private static class MyDialog extends DialogBox {
        public MyDialog() {
            // Set the dialog box's caption.
            setText("My First Dialog");
            // DialogBox is a SimplePanel, so you have to set it's widget
property to
            // whatever you want its contents to be.
            Button ok = new Button("OK");
            ok.addClickListener(new ClickListener() {
                public void onClick(Widget sender) {
                    MyDialog.this.hide();
                }
            });
            setWidget(ok);
        }
    }

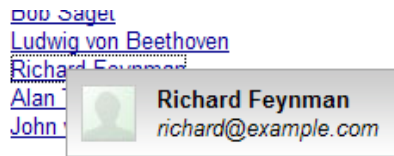
    public void onModuleLoad() {
        Button b = new Button("Click me");
        b.addClickListener(this);
        RootPanel.get().add(b);
    }

    public void onClick(Widget sender) {
        // Instantiate the dialog box and show it.
        new MyDialog().show();
    }
}

```

```
}
```

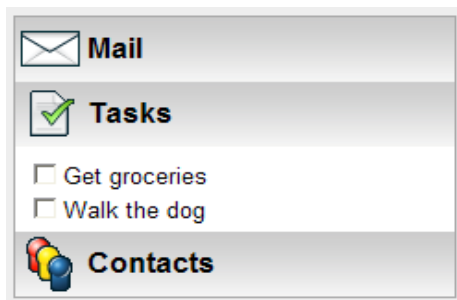
弹出式面板 (PopupPanel)



示例代码:

```
public class PopupPanelExample implements EntryPoint,
ClickListener {
    private static class MyPopup extends PopupPanel {
    public MyPopup() {
        // PopupPanel's constructor takes 'auto-hide' as its
        boolean parameter.
        // If this is set, the panel closes itself automatically
        when the user
        // clicks outside of it.
        super(true);
        // PopupPanel is a SimplePanel, so you have to set it's
        widget property to
        // whatever you want its contents to be.
        setWidget(new Label("Click outside of this popup to
        close it"));
    }
    }
    public void onModuleLoad() {
        Button b = new Button("Click me");
        b.addClickListener(this);
        RootPanel.get().add(b);
    }
    public void onClick(Widget sender) {
        // Instantiate the popup and show it.
        new MyPopup().show();
    }
    }
```

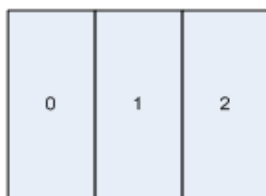
堆栈面板 (StackPanel)



示例代码：

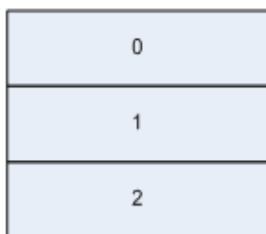
```
public class StackPanelExample implements EntryPoint {
    public void onModuleLoad() {
        // Create a stack panel containing three labels.
        StackPanel panel = new StackPanel();
        panel.add(new Label("Foo"), "foo");
        panel.add(new Label("Bar"), "bar");
        panel.add(new Label("Baz"), "baz");
        // Add it to the root panel.
        RootPanel.get().add(panel);
    }
}
```

✚ 水平面板 (HorizontalPanel)



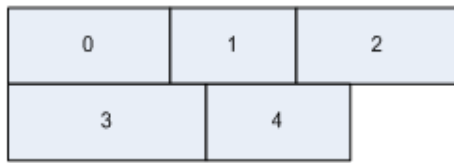
示例代码：

✚ 垂直面板 (VerticalPanel)



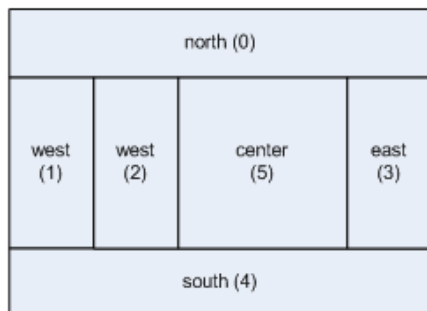
示例代码：

✚ 流面板 (FlowPanel)



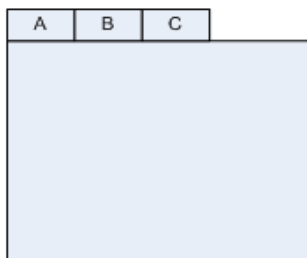
示例代码：

停靠面板 (`DockPanel dock`)



示例代码：

标签页面板 (`TabPanel`)



示例代码：

■ 事件和监听器

界面元素使用著名的监听器模式发出事件。

GWT 中的事件使用“监听器接口”模式，类似于其他用户界面框架。一个监听器接口定义了一个或多个可被界面元素响应事件时调用的方法。一个类如果要接收特定事件，需要实现相关的接口，并且把自身的一个引用传给界面元素来“支持”特定事件集。

例如 `Button` 类，发出单击事件。相关的监听器接口是 `ClickListener`。

```
public void anonClickListenerExample() {

    Button b = new Button("Click Me");

    b.addClickListener(new ClickListener() {

        public void onClick(Widget sender) {

            // handle the click event

        }

    });

}
```

如上例，使用匿名内部类对大量界面元素来说会是低效的。因为它会引起很多监听器类对象的创建。当它们调用一个监听方法时，界面元素把它们的 `this` 指针作为 `sender` 参数。这允许在多个事件发布者中使用单个监听器。这可以更好地使用内存，但是只增加少量代码，如下例所示：

```
public class ListenerExample extends Composite implements
ClickListener {
    private FlowPanel fp = new FlowPanel();
    private Button b1 = new Button("Button 1");
    private Button b2 = new Button("Button 2");
    public ListenerExample() {
        initWidget(fp);
        fp.add(b1);
        fp.add(b2);
        b1.addClickListener(this);
        b2.addClickListener(this);
    }
    public void onClick(Widget sender) {
        if (sender == b1) {
            // handle b1 being clicked
        } else if (sender == b2) {
            // handle b2 being clicked
        }
    }
}
```

```

    }
}
}

```

某些事件接口指定了多个事件。如果你只对其中一些感兴趣，可以使用事件“适配器”的子类。适配器只是简单使用空方法实现特定事件接口，利用它你可以不用实现接口中的每个方法而派生出一个监听器类。

```

public void adapterExample() {
    TextBox t = new TextBox();
    t.addKeyListener(new KeyListenerAdapter() {
        public void onKeyPress(Widget sender, char keyCode, int
modifiers) {
            // handle only this one event
        }
    });
}


```

■ 创建一个自定义的界面元素

完全用 JAVA 代码创建你自己的界面元素。

Composites

Composites 是创建新的界面元素的最有效率的方式。你能够方便地把一组现存的界面元素绑定到一个 **composite**，这个 **composite** 自身就是一个可重用的界面元素。**Composite** 是一个特定的界面元素，它可以包含其他组件（通常，一个面板），但是在行为上，类似于一个被包含界面元素。使用 **composite** 是使用 **Panel** 子类创建复杂界面元素的较好方法，以下是一个如何创建 **composite** 的例子。

 在 JAVA 代码中从代码片断创建。

 使用 JAVASCRIPT

■ 理解布局

理解界面元素如何在面板中放置。

面板在 **GWT** 中类似于其他用户界面库中的对应物。主要的区别是面板使用 **HTML** 元素，诸如 **DIV** 和 **TABLE** 来放置它们的孩子元素。

根面板 (RootPanel)

你遇到的第一个面板是 **RootPanel**。这个面板总是在容器层级的顶层。默认的 **RootPanel** 包装了 **HTML** 文档的 **BODY** 元素，用 **RootPanel.get()** 方法得到。如果你需要得到包装了其他 **HTML** 文档元素的根面板，可以使用 **RootPanel.get(String)** 方法。

✚ 单元面板 (CellPanel)

单元面板是停靠面板 (DockPanel), 水平面板 (HorizontalPanel), 和垂直面板 (VerticalPanel) 的抽象基类。一般这些面板都会把它们子界面元素放到一个“逻辑”单元中。所以这些单元中的子界面元素可以通过 `setCellHorizontalAlignment()` 和 `setCellVerticalAlignment()` 方法来进行排列。单元面板也可以通过 `CellPanel.setCellWidth` 方法和 `CellPanel.setCellHeight` 方法来设置单元格自身的尺寸 (相对于整个面板)。

✚ 其他面板

其他面板包括甲板面板 (DeckPanel), 表格页面 (TabPanel), 流面板 (FlowPanel), HTML 面板 (HTMLPanel), 和 堆栈面板 (StackPanel)。

✚ 尺寸和度量

可以使用 `setWidth()`, `setHeight()` 和 `setSize()` 方法显式地设置界面元素的大小。这些方法的参数是字符串型 (string) 的, 而不是整型 (integer), 原因是它们接受 CSS 标准, 例如像素 (128px), 厘米 (3cm), 和百分比 (100%)。

■ 样式表

界面元素可以很容易地用级联样式表 (CSS) 进行风格化。

GWT 界面元素使用级联样式表定义视觉风格。每个界面元素有一个关联的样式名字, 它绑定到一个 CSS 样式规则。一个界面元素的样式名字是使用 `setStyleName()` 方法来设置的。例如, 按钮 (Button) 有一个默认的名为 `gwt-Button` 的样式。如果想让所有的按钮显示更大的字体, 可以在你的应用程序 CSS 文件中使用下列规则:

```
.gwt-Button { font-size: 150%; }
```

✚ 复杂样式

某些界面元素有稍微复杂的样式。例如菜单条 (MenuBar), 有下列样式:

```
.gwt-MenuBar { the menu bar itself }
.gwt-MenuBar .gwt-MenuItem { menu items }
.gwt-MenuBar .gwt-MenuItem-selected { selected menu items }
```

在这个例子中, 有两个样式规则应用到菜单项。第一个应用到所有的菜单项 (选中的和未选中的), 同时第二个 (有 `-selected` 后缀) 应用到选中的菜单项。一个选中的菜单项的样式名字会被设置成 `"gwt-MenuItem`

`gwt-MenuItem-selected"`, 这两种样式都会用到。最通常的方式是使用

`setStyleName` 来设置基本样式名, 然后用 `addStyleName()` 和

`removeStyleName()` 方法来添加和去除第二个样式名。

CSS 文件

一般地，样式表被放置到你的模块的公共路径（`public path`）部分的某个包中。然后只需在你的主页（`host page`）中引入即可，例如：

```
<link rel="stylesheet" href="mystyles.css" type="text/css">
```

三、远程过程调用（RPC）

一个易于使用的 RPC 机制，用于从服务器通过标准 HTTP 协议发送或接收 JAVA 对象。

GWT 应用程序和传统的 HTML WEB 应用程序一个根本的不同点，就是当它们执行时，GWT 应用程序不需要再获取新的 HTML 页面。因为使用 GWT 技术的页面实际上更像是运行在浏览器中的一个应用程序，不需要再从服务器请求 HTML 来更新用户界面。毕竟，就像所有的客户机 / 服务器应用程序，GWT 应用也需要经常从服务器取得数据。与服务器通过网络进行交互的机制叫做远程方法调用（RPC），有时也被叫做服务器调用。这种机制可以在客户机和服务器之间通过 HTTP 方便地传递 JAVA 对象。

如果使用得当，RPC 可以把你的所有界面逻辑移到客户端，这可以极大地提高执行效率，减少带宽占用，减少 WEB 服务器的负担，并且使用用户体验更加流畅。

服务器端代码经常作为服务（*service*）被客户端调用，所以进行远程过程调用有时也就是调用一个服务。更直白一点，术语服务（*service*）并不是一般意义上的 WEB 服务（*web service*）的概念。实际上，GWT 服务和简单对象访问协议（SOAP）无关。

规范

- RPC 架构图（略）
- 创建服务

如何构建一个服务接口

从一个客户端 JAVA 接口开始，这个接口扩展了 `RemoteService` 接口。

```
public interface MyService extends RemoteService {
    public String myMethod(String s);
}
```

这个同步接口是你的服务的规范的确定版本。任何这个服务的服务器端实现必须扩展 `RemoteServiceServlet` 并实现这个服务接口。

```
public class MyServiceImpl extends RemoteServiceServlet
implements
    MyService {
    public String myMethod(String s) {
        // Do something interesting with 's' here on the server.
        return s;
    }
}
```

■ 异步接口

在你可以从客户端进行实际远程调用之前，你必须创建另外一个基于你最初服务接口的接口，一个异步接口。例子继续如下：

```
interface MyServiceAsync {
    public void myMethod(String s, AsyncCallback callback);
}
```

异步方法调用的特性需要调用者传入一个回调对象，这个回调对象在异步调用结束时被唤醒，因为根据定义调用结束时调用者不能被阻塞。同样原因，异步方法没有返回值，它们的返回值总是为空。在一个异步调用之后，所有的返回到调用者的通信是通过传回的回调对象进行的。

服务接口和它的异步对应物之间的关系如下：

✚ 如果一个服务接口叫做 `com.example.cal.client.SpellingService`，那么对应的异步接口必须叫做 `com.example.cal.client.SpellingServiceAsync`。这个异步接口必须在同一个包，并且有同样的名字，但是这个名字以 `Async` 作后缀。

✚ 对于你服务接口中的每个方法，

```
public ReturnTye methodName(ParamType1 param1, ParamType2
param2);
```

应该定义一个异步的兄弟方法，如下所示：

```
public void methodName (ParamType1 param1, ParamType2 param2,
AsyncCallback callback);
```

- 实现服务

以一个 **SERVLET** 的方式实现你的服务。

每个服务最终需要实现某些处理，以便响应客户端的请求。服务实现中的服务端处理，是基于著名的 **SERVLET** 架构。

一个服务实现必须扩展 `RemoteServiceServlet`，并且必须实现相关的服务接口。注意：服务实现并不实现服务接口的异步版本。

每一个服务实现都最终是一个 **SERVLET**，但不是扩展了 `HttpServlet`，而是 `RemoteServiceServlet`。`RemoteServiceServlet` 自动处理序列化并且调用你的服务实现中的对应方法。

- 在开发时期测试应用

为了自动装载你的服务实现，在你的模块 **XML** 中使用 `<servlet>` 标志。GWT 开发命令环境包含一个嵌入版本的 Tomcat，它作为开发时进行测试的 **SERVLET** 容器。

- 把服务部署到产品中

在生产中，你能够使用任何适合你的应用程序的 **SERVLET** 容器。你只需要确定客户端代码可以通过在 `web.xml` 中映射的 **SERVLET** 的 URL 进行服务调用。

- 进行一次实际调用

如何从客户端进行一次实际的远程过程调用。

从客户端进行一次 **RPC** 调用的过程，总有相同的步骤。

1. 使用 `GWT.create()` 方法实例化服务接口。
2. 为服务代理使用 `ServiceDefTarget` 指定服务入口 URL
3. 创建一个异步回调类对象，以便 RPC 调用结束后使用。
4. 发出一个调用。

■ 例子

假定你想调用一个服务接口中的方法，如下定义

```
public interface MyEmailService extends RemoteService {  
    void emptyMyInbox(String username, String password);  
}
```

它的异步对应接口会如下所示：

```
public interface MyEmailServiceAsync {  
    void emptyMyInbox(String username, String password,  
        AsyncCallback callback);  
}
```

客户端调用如下所示：

```
public void menuCommandEmptyInbox() {  
    // (1) Create the client proxy. Note that although you are creating  
    the  
    // service interface proper, you cast the result to the asynchronous  
    // version of  
    // the interface. The cast is always safe because the generated proxy  
    // implements the asynchronous interface automatically.  
    //  
    MyEmailServiceAsync emailService = (MyEmailServiceAsync)  
GWT.create(MyEmailService.class);  
    // (2) Specify the URL at which our service implementation is running.  
    // Note that the target URL must reside on the same domain and port  
    from  
    // which the host page was served.  
    //  
    ServiceDefTarget endpoint = (ServiceDefTarget) emailService;  
    String moduleRelativeURL = GWT.getModuleBaseURL() + "email";  
    endpoint.setServiceEntryPoint(moduleRelativeURL);  
}
```

```

// (3) Create an asynchronous callback to handle the result.
//
AsyncCallback callback = new AsyncCallback() {
public void onSuccess(Object result) {
    // do some UI stuff to show success
}
public void onFailure(Throwable caught) {
    // do some UI stuff to show failure
}
};
// (4) Make the call. Control flow will continue immediately and later
// 'callback' will be invoked when the RPC completes.
//
emailService.emptyMyInbox(fUsername, fPassword, callback);
}

```

缓存服务代理，来避免在后续调用中重新创建是安全的。

● 序列化类型

使用 GWT 的自动序列化。

方法参数和返回值必须是可被序列化的，这意味着它们必须符合某些限制。GWT 尽力使序列化直白，符合下列规定，则一个类型是序列化的并可以用于服务接口。

- ✓ 是原生类型，例如：char, byte, short, int, long, boolean, float, 或 double
- ✓ 是 String, Date 或一个原生类型的包装类，例如：Character, Byte, Short, Integer, Long, Boolean, Float, 或 Double。
- ✓ 是一个序列化类型的数组（包括其他序列化数组）
- ✓ 是一个序列化的用户自定义类，或者
- ✓ 有至少一个序列化的子类。

■ 序列化的用户自定义类

如何符合下列条件，一个用户自定义类则是可序列化的。

1.它是直接或间接派生自 `IsSerializable`，并且

2.所有的非瞬时字段都是序列化的。

瞬间 (Transient) 字段在 RPC 时不会被交换。被声明为 `final` 的字段在 RPC 阶段也不可交换，则它们一般应该被标记为 `transient`

■ 多态

GWT RPC 支持多态参数和返回类型。为了更好地使用多态，在定义服务接口时，你应该尽可能详细你的设计。精确性越高，编译器就可以更好地工作，所以在优化程序时要去掉不必要的代码。

■ 类型变量

集合类，例如： `java.util.Set` 和 `java.util.List` 不好掌握，因为它们操纵 `Object` 类对象。为了使集合序列化，你应该指定需要被放置的对象的精确类型。这需要使用 JAVADOC 声明 (annotation) `@gwt.typeArgs`。为集合 (collection) 定义一个条目类型意味着你会确保这个集合只能放置指定的类型或是它的派生类。这有助于生成高效的代码。违反这个规则会引起不可预知的行为。

在一个序列化的用户定义类中声明集合类型字段

```
public class MyClass implements IsSerializable {
    /**
     * This field is a Set that must always contain Strings.
     * @gwt.typeArgs <java.lang.String>
     */
    public Set setOfStrings;
    /**
     * This field is a Map that must always contain Strings as
     its keys and
     * values.
     * @gwt.typeArgs <java.lang.String,java.lang.String>
     */
    public Map mapOfStringToString;
}
```

注意这里不需要在`@gwt.typeArgs` 声明中指定字段的名字，因为可以推断出来。

同样地， 可以标注参数和返回值：

```
public interface MyService extends RemoteService {  
    /**  
     * The first annotation indicates that the parameter named  
     * 'c' is a List  
     * that will only contain Integer objects. The second  
     * annotation  
     * indicates that the returned List will only contain String  
     * objects  
     * (notice there is no need for a name, since it is a return  
     * value).  
     *  
     * @gwt.typeArgs c <java.lang.Integer>  
     * @gwt.typeArgs <java.lang.String>  
     */  
    List reverseListAndConvertToStrings(List c);  
}
```

注意参数的标注必须包含参数的名字除了集合项目类型，返回值标注不需要。

小提示：

虽然此术语非常简单，GWT “序列化” 的概念和标准的 JAVA 接口 `Serializable` 的序列化不同。

● 处理异常

处理由于调用失败或服务抛出的异常。

进行（远程过程调用）RPC 时会引发各种各样的错误。网络失败，服务器崩溃，等等。GWT 让你可以用 JAVA 异常的方式处理这些情况。RPC 相关的异常可分为两类。

■ 受查异常

服务接口方法支持 `throws` 声明，指出哪种异常可能会被从服务实现抛回客户端。

调用者应该实现 `AsyncCallback.onFailure(Throwable)` 来检查在服务接口中指定的异常。

■ 未预期异常

一个 **RPC** 可能最终没有到达服务实现。可以有很多原因：网络断掉，**DNS** 服务器不可用，**HTTP** 服务器没有监听等等。这种情况下，一个 `InvocationException` 被传送到你的 `AsyncCallback.onFailure(Throwable)` 实现。这个类叫作 `InvocationException` 是因为问题是关于试图调用而不是服务实现。

一个 **RPC** 也可能因为调用异常而失败，如果调用没有到达服务器。但是一个未声明的异常会发生在调用的一般处理时。有很多理由：一个必须的服务器资源，例如数据库，可能不可用，一个 `NullPointerException` 可能会被抛出，是因为服务实现的 **BUG**，等等。在这些情况下，一个 `InvocationException` 会在应用代码中抛出。

● 开始使用异步调用

异步调用开始时可能是可笑的，但是最终你的用户会感谢你。

例如：假定你的应用程序显示一个包含很多界面元素的大的表格 (`table`)。构建并部署所有的界面元素是很耗时的。同时，你需要从服务器取回数据在表格中显示。这是一个使用异步调用的完美的理由。在你开始构建你的表格和它的界面元素之前，发动一个异步调用去请求数据。同时服务器正在取得被请求的数据，浏览器正在执行你的用户界面代码。当客户端最终收到服务器端数据时，表格被构建并布局，数据也可以显示了。

为了说明这种技术如何高效，假定构建表格需要 1 秒，取数据需要 1 秒。如果你同步调用服务器，全部过程最少需要 2 秒。但是如果你异步地取回数据，全部过程只要 1 秒，虽然你做了 2 秒的事。

使用异步调用最难的事是这种调用是非阻塞的。毕竟，**JAVA** 内部类对此实现也是较难的。

小提示：

`AsyncCallback` 接口是你处理 **RPC** 响应的关键接口。

- 架构视图

比较几种实现服务的方法。

有各种方式在你的应用架构中得到服务。第一要理解的是 **GWT** 服务不是取代 **J2EE** 服务的，也不是为了提供公司 **WEB** 服务。基本上，**GWT RPC** 只是一个简单的“从客户机到服务器”的方法。换句话说，你使用 **RPC** 完成任何是你的应用程序的一部分，但是不能客户机上完成。

从构架上来说，你能通过两种可互换的方式使用 **RPC**。不同只在于你的应用程序的爱好如何。

第一个和最直接的方式关于服务定义，是把它们看作你的应用程序的全部后端。从这个观点出发，客户端代码是你的“前端”，并且所有的运行在服务器端的服务代码是“后端”。如果你用这种方式，你的服务实现更像是通用的 **API**，而不是和特定的应用程序绑定紧密。你的服务定义可能通过 **JDBC** 或 **Hibernate** 或服务器文件系统的文件，直接访问数据库。对于很多应用，这种观点是正确的，它可能是高效的，因为它有较少的调用层数。

在更复杂、多层构架中，你的 **GWT** 服务定义只是轻量级的门户，它通过后端服务器环境，例如 **J2EE** 服务器。从这个观点来说，你的服务可以被看作“半服务器”式的应用程序用户接口。不像通用服务，这些服务专用于你的用户界面。

这种构架是适合的，如果你想让你的后端服务运行在一个物理上独立的计算机上上的 **HTTP** 服务器上。

四、JUnit 集成

和 **JUnit** 的集成使测试你的 **AJAX** 代码和测试 **JAVA** 代码一样容易。

GWT 包含了一个特殊基类 **GWTTestCase**，它提供了 **JUnit** 集成。在 **JUnit** 中运行一个编译后的 **GWTTestCase** 子类将启动一个不可见的 **GWT** 浏览器。

默认情况下，运行在主机模式下的测试，类似于 **JVM** 中的一般 **JAVA** 字节码。覆写这些默认为需要给 **GWT** 开发环境传递参数。参数不能直接通过命令行传递，因为一般的命令行参数直接到达 **JUnit** 运行器。而是通过定义系统属性 **gwt.args** 来把参数传递到 **GWT**。

例如：在 **WEB** 模式中运行，声明 **-Dgwt.args="-web"** 作为 **JVM** 参数。。。。

- 创建一个测试用例 (Test Case)

GWT 包括一个便利工具 `junitCreator`，它可以生成测试用例，在主机模式和 WEB 模式添加测试脚本。手工创建它的步骤如下：

1. 定义一个扩展自 `GWTTestCase` 的类。
2. 创建一个模块，包含你的测试用例。如果你正在向一个已存在的 GWT 应用添加测试用例，通常你能够使用已存在模块。
3. 实现方法 `GWTTestCase.getModuleName()` 来返回模块的完整名字。
4. 编译你的测试用例到字节码（使用 `JAVAC` 或 `JAVA IDE`）。
5. 当运行测试用例，确定你的类路径包含：
 - a) 你的项目的 `src` 目录。
 - b) 你的项目的 `bin` 目录。
 - c) `gwt-user.jar`
 - d) `gwt-dev-windows.jar` (或 `gwt-dev-linux.jar`)
 - e) `junit.jar`

- 例子

编写 `com.example.foo.client.FooTest` 测试用例。

```
public class FooTest extends GWTTestCase {
    /*
     * Specifies a module to use when running this test case. The returned
     * module must cause the source for this class to be included.
     *
     * @see com.google.gwt.junit.client.GWTTestCase#getModuleName()
     */
    public String getModuleName() {
        return "com.example.foo.Foo";
    }
    public void testStuff() {
        assertTrue(2 + 2 == 4);
    }
}
```

创建 `com.example.foo.Foo` 模块

```
<module>
  <!-- Module com.example.foo.Foo -->
  <!-- Standard inherit. -->
  <inherits name='com.google.gwt.user.User' />
  <!-- implicitly includes com.example.foo.client package -->
  <!-- OPTIONAL STUFF FOLLOWS -->
  <!-- It's okay for your module to declare an entry point. -->
  <!-- This gets ignored when running under JUnit. -->
  <entry-point class='com.example.foo.FooModule' />
  <!-- You can also test remote services during a JUnit run. -->
  <servlet path='/foo'
class='com.example.foo.server.FooServiceImpl' />
</module>
```

小提示:

你不需要为每个测试用例创建单独的模块。在上例中，一些 `com.example.foo.client`（或子包）中的测试用例能够共享 `com.example.foo.Foo` 模块。

五、国际化（Internationalization）

用单代码基础支持多本地（`locale`）

待定未完

GWT 包括了一系列工具来国际化你的应用程序和库。GWT 国际化支持提供了各种技术来处理字符串、值和类。

- 起步

因为 GWT 支持多种方式来国际化你的代码，你应该首先确定哪种方式最适合你的开发需求。

你是从草稿写代码么？（Are you writing code from scratch?）

如果是，你可能需要使用静态字符串国际化（[static string internationalization](#)）技术。

你想要国际化大部分设置或最终用户信息么？

如果你有很多设置项目（一般你会使用属性文件），考虑使用常量（[Constants](#)）。如果你有很多最终用户信息，那么可能需要使用信息（[Messages](#)）。

你有现存的本地属性文件，并用想要重用么？

[i18nCreator tool](#) 能够自动生成扩展自 [Constants](#) 或 [Messages](#) 的接口。

你想把 GWT 功能加入到一个已经实现国际化的 WEB 应用程序中么？

[Dictionary](#) 会让你不必使用 [GWT's concept of locale](#) 来与已存在页面交互。

你只是想用一种简单的方式在客户端使用属性文件，而不考虑国际化？

你不用指定地点（locale），只使用常量（[Constants](#)）即可。

- 国际化技术

在 GWT 中，有多种国际化技术可供选择。以迎合开发者所需的最大可伸缩性、效率、可维护性和互操作性。

静态字符串国际化（[Static string internationalization](#)）是一系列高效且类型安全的技术，它依靠强类型的 JAVA 接口，属性文件（[properties files](#)），和代码生成来提供地点敏感的信息和配置项。这些技术依靠 [Constants](#) 和 [Messages](#) 接口。

另一方面，动态字符串国际化（[dynamic string internationalization](#)）是一个过分单纯化且有弹性的技术，用于查找在模块主页中的本地化的值，而不需要重新编译应用程序。这项技术由类 [Dictionary](#) 支持。

使用类似于静态字符串国际化的方法，GWT 也提供使用位置敏感国际化技术。这是一种高级技术，你可能不需要直接使用，虽然它对实现复杂国际化库是有用的。

- I18N 模块

和国际化有关的核心类型都在 `com.google.gwt.i18n` 包里：

- Constants

用于本地化常量值。

- Messages

用于本地化需要参数的信息

- ConstantsWithLookup

类似常量，但是有附加的查找伸缩性，用于高度数据驱动的应用程序。

- Dictionary

用于向现存本地化 WEB 页添加 GWT 模块

- Localizable

用于把国际化规则封装入一个类。

GWT 国际化类型包含在模块 `com.google.gwt.i18n.I18N`。要使用这些类型，你的模块必须继承它

```
<module>
```

```
<!-- other inherited modules, such as com.google.gwt.user.User -->
```



```
<inherits name="com.google.gwt.i18n.I18N"/>

<!-- additional module settings -->

</module>
```

- 规定

- 静态字符串国际化 (Static String Internationalization)

一个类型安全，并经过优化的方法，用来国际化字符串。

- 动态字符串国际化 (Dynamic String Internationalization)

一个简单的国际化字符串方法，不支持 GWT `locale` 客户端属性。

- 指定一个地点 (Locale)

如何在部署时添加地点和指定地点 (`locale`) 客户端属性。

- 本地化属性文件

如何使用 [Constants](#) 或 [Messages](#) 创建属性文件。

六、JavaScript 本地接口 (JSNI) (未完成，不提倡使用)

把手写 JavaScript 代码混合进你的 JAVA 类，以调用底层的浏览器功能。

待定

GWT 编译器把 JAVA 源代码翻译成 JAVASCRIPT。有时，把手写的 JAVASCRIPT 代码混合入你的 JAVA 源代码也是有用的。例如：某些核心 GWT

类最底层的功能是用 JAVASCRIPT 写的。GWT 从 JAVA 本地接口 (JNI) 概念，实现了 JAVASCRIPT 本地接口 (JSNI)。

写 JSNI 方法是一个强大的技术，但是不应经常使用。JSNI 代码不易跨浏览器使用，更容易引起内存泄漏，和 JAVA 工具配合不好，不易编译与优化。

我们认为 JSNI 是行内装配代码的 WEB 等价物，你能够：

- ✚ 在 JAVASCRIPT 中直接实现 JAVA 方法
- ✚ 在已存在的 JAVASCRIPT 代码上包装类型安全的 JAVA 方法签名
- ✚ 从 JAVASCRIPT 调用 JAVA 代码，或相反。
- ✚ 通过 JAVA/JAVASCRIPT 边界抛出异常。
- ✚ 从 JAVASCRIPT 读写 JAVA 属性
- ✚ 使用主机模式调试 JAVA 代码（用 JAVA 调试器）和 JAVASCRIPT 代码（用 JAVASCRIPT 调试器，现只在 WINDOWS 中可用）。

小提示：

当从 JSNI 访问浏览器的 WINDOW 和 DOCUMENT 对象时，你必须用 `$wnd` 和 `$doc` 分别引用它们。你的编译过的脚本运行在一个嵌套的框架 (frame) 中，`$wnd` 和 `$doc` 被自动初始化来正确地引用到主页中的 WINDOW 和 DOCUMENT。

● 规定

✚ 写本地 JAVASCRIPT 方法

声明一个本地方法且把方法体写入特定格式的注释。

JSNI 方法以 `native` 来声明，并把 JAVASCRIPT 代码包含入一个特定格式的注释块，在参数列表末尾和最后的分号之间。一个 JSNI 注释块以 `/*-{` 开头，以 `}-*/` 结尾。JSNI 方法的调用类似于一般 JAVA 的调用。它们可以是静态或实例化方法。

- 例子

```
public static native void alert(String msg) /*-{  
    $wnd.alert(msg);  
}-*/;
```

从 JAVASCRIPT 访问 JAVA 方法和属性

手写 JAVASCRIPT 能够访问 JAVA 对象的方法和属性。

从 JSNI 方法的 JAVASCRIPT 实现去操作 JAVA 对象是有用的，它需要特殊语法支持。

- 从 JAVASCRIPT 调用 JAVA 方法

它类似于从 C 代码在 JNI 中调用 JAVA 代码。不同的是，

 在 JAVA 源代码和 JAVASCRIPT 源代码中共享对象。

JAVA 对象如何在 JAVASCRIPT 代码中出现，或相反的情况。

异常和 JSNI

JAVASCRIPT 异常和 JAVA 异常的互动。

七、GWT 应用程序自定义应用程序框架

在开始开发一个 GWT 应用程序时，我们已经有了一个比较稳定的程序框架。类似于其他程序框架，遵守这个框架开发程序，可以避免在基础性问题上面占用过多的时间和精力，从而使开发人员可以更好地从全局上把握程序。其他优势不再赘述。

框架规定


- 程序入口类必须继承 `pharos.web.core.client.PharosEndPoint`
- 程序入口类在开始处定义本模块程序中所用到的常量，常量必须初始化。示例如下：


```
private final String CON_FILTER="";
```


- 在程序入口类常量定义的下方，定义自定义的全局容器，示例如下：

```
private Context context=null;
```

- 自定义全局容器（Context）有关规定

 类名必须是 Context 或以 Context 结尾。

 这个类必须继承 ContextBase 类。

 这个类是一个 JAVA BEAN，提供了对其属性的 GET 和 SET 方法。

- ✚ 其属性是程序中需要引用的对象，主要是界面元素，也要有一个得到自身对象的方法。
- ✚ 属性的实例化采用单例懒惰加载方式，示例代码如下：

```
public TreeHelp getTreeHelp() {  
    if (treeHelp == null) {  
        treeHelp = new TreeHelp();  
    }  
    return treeHelp;  
}
```

- ✚ 这个类是由开发者定制开发的。
- 下一步编写 `onModuleLoad()` 方法
 - ✚ 取得全局容器对象。样板代码：`context=Context.getInstance()`，并设置给父类，如：`setContextBase(context)`；
 - ✚ 设置 布局类实例 `this.setTemplateLayout(new TemplateDockNoMenu())`；布局类的作用是：进行界面整体性的布局，即形成界面上的相对静态的部分，这部分工作和上一步中的工作相加，组成了完整的界面布局定义。
 - ✚ 先调用超类的 `onModuleLoad()` 方法。样板代码：`super.onModuleLoad()`；
- 入口程序类实现 `loadBusinessWidget()` 方法：
 - ✚ 对全局对象中的属性进行组装，主要是把各种界面元素放入用来布局的各种面板。相当于对界面进行初始化。这一步骤不是必须的，这部分工作也可以放到业务组件类的构造函数中去做。
 - ✚
- 布局类必须继承 `Composite` 类并实现 `TemplateLayout` 接口。在其 `paint()` 方法中，开发人员编写代码，进行界面的整体布局。
- 业务组件类，它是界面布局类的 `paint()` 方法操作的对象。业务组件类的规定如下：
 - ✚ 继承 `Composite` 类，使其成为一个界面组件。
 - ✚ 实现 `pharos.web.core.controller.SysnetFace` 接口，该接口方法说明如下：
 - `init()` 方法：用业务数据模型初始化或刷新界面。
 - `bind()` 方法：把业务组件类中的值绑定到业务数据模型。
 - `hide()` 方法：业务组件不显示。
 - `show()` 方法：业务组件显示。
 - `clear()` 方法：清空所有子元素。
 - `reset()` 方法：把所有子元素值置为初始状态。
 - ✚ 在业务组件类一开始，声明本业务组件内的界面元素。

样板代码：

```
...  
private TextBox textBoxProposalNum=new TextBox();  
private TextBox textBoxPolicyNum=new TextBox();
```

```
private TextBox textBoxDocumente=new TextBox();
```

```
...
```

- ✚ 接着，声明本业务组件类所用的业务数据模型，该业务数据模型是一个 JAVA BEAN，其属性是业务组件中需要显示和保存的数值。
- ✚ 在构造函数中，设置界面元素的可见性，并把分散的界面元素整合为一个整体
- ✚ 在业务组件类中编写 `init()` 方法，此方法以一个业务数据模型为参数，作用是把数据模型中的数据赋给业务组件中的相应界面元素。此方法是在初始化或刷新界面时调用。

样板代码：

```
...
```

```
textExchange.setText(model.getExchange());
```

```
textTax.setText(model.getTax());
```

```
textRenewal.setText(model.getRenewal());
```

```
...
```

- ✚ 在业务组件类中编写 `bind()` 方法，此方法没有参数与返回值，作用是收集业务模型中的值，把这些值集中到全局变量业务数据模型对象中。此方法是在保存数据时调用。

样板代码：

```
...
```

```
model.setBeneficiary(textBoxBeneficiary.getText());
```

```
model.setPayer(textBoxBeneficiary.getText());
```

```
model.setAgent(textBoxAgent.getText());
```

```
...
```

- 对业务数据模型的有关规定

- ✚ 必须继承 `pharos.web.core.model.BaseModel` 类。

样板代码：

```
...
```

```
public class MainInformationModel extends BaseModel{
```

```
...
```

- ✚ 业务数据模型类必须有一个空的构造函数。

样板代码：

```
...
```

```
public MainInformationModel(){}  
...
```

- ✚ 业务数据模型类只能使用 `JAVA.LANG.*` 下的数据类型（不包含下层子包）和 `JAVA.UTIL.*` 下的数据类型（不包含下层子包）。不使用 `LONG` 型，而是使用 `INT` 型。

- 对客户端服务接口的规定

- ✚ 接口中的方法必须抛出 `pharos.web.core.exception.WebRequestException` 异常。
模板代码:

```
public fetchFilter(ProFilter filter) throws WebRequestException;
```

- 对服务端代码的规定
 - ✚ 服务器端的服务实现类，有一个对应的具体业务实现类，即 ACTION 类。这两个类中的方法一一对应，方法签名完全相同，由服务实现类中的方法调用 ACTION 类中的方法。

模板代码:

```
public TreeItems fetPanTree(String proId) throws WebRequestException{
    NewProposalAction action=new NewProposalAction();
    return action. fetPanTree(proId,this.getThreadLocalRequest());
}
```

八、自定义组件部分

公共界面组件使用说明

- 1 页面整体布局
 - 1.1 基类

基本说明	示例见TEST工程 基类 Dock布局 北 (logo,menu)，西 (Stack抽屉)，中1 (help)，中 (work:Tab,toolPanel)，南 (MessageWindow)
使用方法	作为GWT程序入口，运行后生成内容为空的布局框架。 各个框架容器都有get/set方法。
类路径	<code>pharos.web.core.client.TemplateDockSuperEndPoint</code>

- 1.2 抽屉

基本说明	示例见TEST工程 抽屉公共类，专用组件
------	-----------------------------

使用方法	
类路径	pharos.web.core.client.LeftStack

1.3 多页Tab

基本说明	<p>示例见TEST工程</p> <p>专用组件</p>
使用方法	
类路径	pharos.web.core.client.WorkspaceTab

1.4 消息窗口

基本说明	<p>示例见TEST工程</p> <p>用于在屏幕上显示服务器返回的提示信息。</p>
使用方法	<p>消息处理框</p> <p>功能：</p> <p>(1) 显示的消息分类： error msg warning msg 显示 operator msg</p> <p>(2) 不同类可以放在不同的 Tab Page ， 或者不同的颜色。</p> <p>(3) 可以最大化，可以缩小。</p> <p>(4) 提供外部调用接口： 增加消息， 清空窗口</p> <p>示例代码：</p> <pre>... rootPanel = RootPanel.get(); MessageWindow mw=new MessageWindow(); mw.add("aaa"); mw.add("bbb"); mw.add("ccc"); rootPanel.add(mw); ...</pre>
类路径	pharos.web.core.widget.MessageWindow

1.5 下拉菜单

基本说明	<p>示例见TEST工程</p> <p>专用组件</p>
使用方法	
类路径	pharos.web.core.client.TopMenu

1.6 上下文对象容器基类

基本说明	
使用方法	<p>对象容器。 对象区分：容器类（如：抽屉、Tab），Widget类（各业务模块定义的）</p> <p>ContextBase只负责定义和管理 容器类 的对象。</p> <p>负责：</p> <p>（1）存放所有需要交互的 大控件 。</p> <p>（2）管理实例（初始化）</p>
类路径	pharos.web.core.client.ContextBase

2 对话框

2.1 对话框扩展类

基本说明	未知
使用方法	
类路径	pharos.web.core.widget.DialogBoxEx

2.2 版本信息对话框

基本说明	用于用户点击“帮助”→“关于”菜单项后，显示系统版本信息。暂时不支持参数化版本信息内容。
使用方法	<p>示例代码：</p> <pre> ... AboutDialog dlg = new AboutDialog(); int left = [此对象屏幕横坐标]; int top = [此对象屏幕纵坐标]; dlg.setPopupPosition(left, top); dlg.show(); ... </pre>
类路径	pharos.web.core.client.AboutDialog

2.3 错误信息对话框

基本说明	在后台服务调用失败后，在屏幕上显示此对象。
使用方法	<p>示例代码：</p> <p>注：e为WebRequestException类型的对象。</p> <pre> ... ErrorMsgDialog dialog = new ErrorMsgDialog(e); int left = 200; int top = 200; dialog.setPopupPosition(left, top); dialog.setSize("400", "200"); dialog.show(); ... </pre>
类路径	pharos.web.core.widget.ErrorMsgDialog

2.4 进度条对话框

基本说明	在浏览器端调用服务端服务时，在前台显示一个进度条，同时屏幕不可操作，直到服务器端返回结果。
使用方法	<p>在浏览器端调用服务端服务之前，在屏幕上显示进度条。在服务器返回结果后，去除屏幕上的进度条。</p> <p>示例代码：</p> <p>一、在事件监听器中：</p> <p>...</p> <pre>ProgressDialog.getInstance().addProgress("[进度条显示名]");</pre> <p>[调用服务器端服务]</p> <p>二、在后台服务回调函数中：</p> <p>...</p> <pre>ProgressDialog.getInstance().removeProgress();</pre> <p>...</p>
类路径	pharos.web.core.widget.ProgressDialog

3 输入组件

3.1 日期输入

基本说明	扩展TextBox，提供日历选择
使用方法	直接生成界面对象即可。
类路径	pharos.web.core.widget.DateTextBox

3.2 数字输入

基本说明	<p>扩展TextBox，只能录入 0-9</p> <p>发现的BUG：</p> <p>1、用小键盘输入数字，弹出输入错误提示框</p> <p>2、输入数字后，删除输入的数字，再输入字母，不提示输入错误。</p>
使用方法	直接生成界面对象即可。
类路径	pharos.web.core.widget.IntegerTextBox

3.3 多类型录入组件

基本说明	依赖 ColumnItem 定义
使用方法	
类路径	<p>pharos.web.core.widget.DataInputBox 控件</p> <p>pharos.web.core.model.ColumnItem 数据模型</p>

3.4 多列通用录入控制器

基本说明	依赖 ColumnItem 定义,支持显示多少列
使用方法	
类路径	pharos.web.core.widget.DataEntries 控件 pharos.web.core.model.ColumnItem 数据模型

4 异常组件

4.1 异常类

基本说明	必须自定义, 否则后台的异常传递不到前台
使用方法	
类路径	pharos.web.core.exception.WebRequestException

4.2 异常处理器

基本说明	使用 错误信息对话框 来显示
使用方法	
类路径	pharos.web.core.exception.ExceptionHandle

5 其他组件

5.1 树

基本说明	见TEST工程
使用方法	
类路径	pharos.web.core.widget.SysTree 控件 pharos.web.core.model.TreeItems 数据

5.2 Table

基本说明	不完善
使用方法	
类路径	pharos.web.core.widget.Table 控件 pharos.web.core.model.TreeItems 数据

5.3 ListBox

基本说明	扩展, 支持鼠标双击
使用方法	
类路径	pharos.web.core.widget.SysnetListBox 控件

5.4 回调类

基本说明	统一处理与后台交互的错误
使用方法	
类路径	pharos.web.core.widget.SysnetAsyncCallback

6 UI行为接口 自定义业务组件实现该接口

6.1 Action基类

基本说明	规范 Action的编写
使用方法	
类路径	pharos.web.core.action.BaseAction

6.2 Model基类

基本说明	规范 Model (前台与后台交互用的) 的编写
使用方法	
类路径	pharos.web.core.model.BaseModel

6.3 组合类基类

基本说明	规范 自定义组件 (业务功能) Composite 的编写
使用方法	
类路径	pharos.web.core.widget.SysnetFace

6.4 测试类基类

基本说明	规范 自定义组件的测试 Composite 的编写
使用方法	
类路径	pharos.web.coretest.model.TestFace