

# Qml应用性能优化

# QML的执行

- QML的编译
- QML的执行、实例化
- QML的渲染

setSource

App.exec

QML的编译

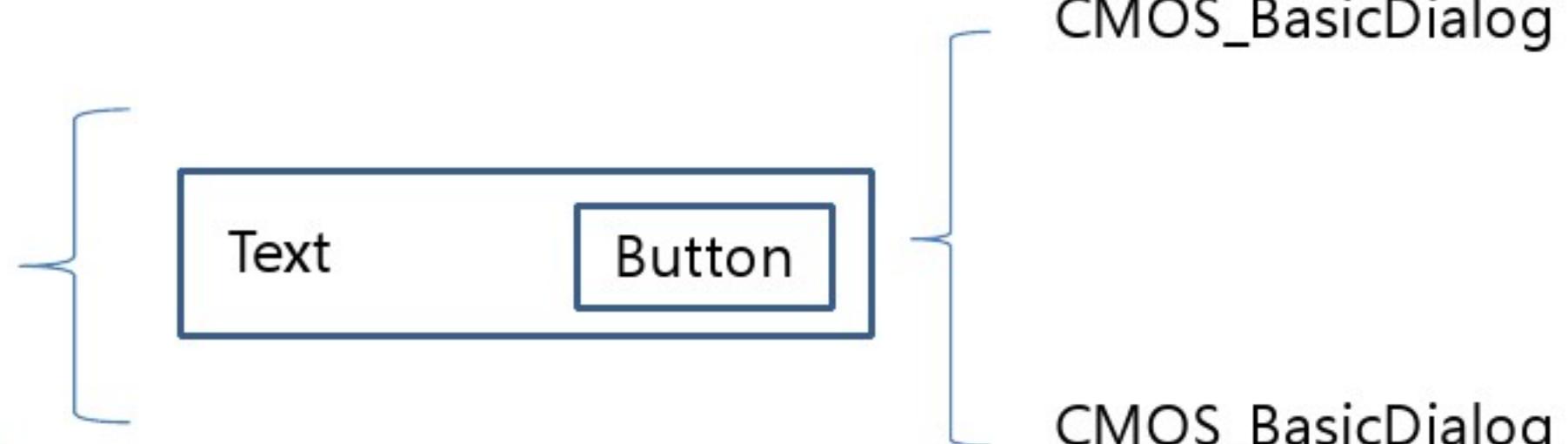
QML的执行、实  
例化

QML的显示

# QML的设计思想

- QML应用在启动时，会对qml进行编译，并且对所有元素进行实例化
  - 比如显示一个Button : id.visible=true
- QML不是一个传统的解释型语言，运行时
  - 将所有代码编译
  - 将所有代码实例化
  - 目的：牺牲启动性能，保证应用的运行性能
- QML的这种设计，导致了。。

# QML性能分析



应用在启动的时候，会创建14个CMOS\_BasicDialog  
,每个CMOS\_BasicDialog用时20ms，总共花费280ms

List

# qmlprofiler

- 需要在应用的.pro开启qml debug：  
`CONFIG += qml_debug`
- 编辑appconfig.xml，修改应用的启动项  
`/usr/bin/cmoscalculator -qmljsdebugger=port:3768,block`
- 重启机器
- 点击运行应用
- 在PC端运行QtCreator

# QML的编译

- 所有通过import方式引入进来的qml代码都将会被编译
- 一个qml文件只会被编译一次
- Qml文件会被编译成为特定的字节码

# 控制QML的编译

- 将QML文件分到多个不同的QML文件
- 采用动态创建QML组件的方式

# QML实例化

- 所有元素在运行前都会被实例化
  - 内存占用很高
  - 应用的启动速度很慢
- 优化策略：用到的时候才需要实例化
  - 采用Qt.createComponent()的方式，动态创建qml组件

# 三种不同的写法

- 不编译、不实例化

```
Qt.createComponent( "qrc:/qml/Alarm/AlarmMain.qml" )
```

- 只编译，不实例化

```
Component {
    Item {
        CMOS_PageStackWindow { }
        CMOS_BasicDialog { }
        CMOS_LineEdit { }
    }
}
```

- 编译且实例化

```
Tab {
    id : alarm
}
```

# 异步Loader

- 本身并不节省时间
- 可以增加并行效果
- 可以在总时间不变的情况下，提高用户体验

# 性能优化

- Do it faster
- Do it in parallel
- Do it later
- Don't do it at all
- Do it before

# QML的优化技巧

- Text Element
- Image
- Controlling Element Lifetime
- Render

[qt-project.org/doc/qt-5/qtquick-performance.html](http://qt-project.org/doc/qt-5/qtquick-performance.html)

# Text Element

- PlainText
- StyledText
- AutoText
- RichText



# Images

- **Asynchronous Loading**
  - Images are often quite large, and so it is wise to ensure that loading an image doesn't block the UI thread. Set the "asynchronous" property of the QML Image element to true to enable asynchronous loading of images from the local file system
- **Explicit Source Size**
  - If your application loads a large image but displays it in a small-sized element, set the "sourceSize" property to the size of the element being rendered to ensure that the smaller-scaled version of the image is kept in memory, rather than the large one.

# Controlling Element Lifetime

- **Lazy Initialization**
  - there is no better way to reduce startup time than to avoid doing work you don't need to do, and delaying the work until it is necessary.
- **Using Loader**
  - Using the "active" property of a Loader, initialization can be delayed until required.
  - Using the overloaded version of the "setSource()" function, initial property values can be supplied.
  - Setting the Loader asynchronous property to true may also improve fluidity while a component is instantiated.

# Controlling Element Lifetime

- **Using Dynamic Creation**
  - Developers can use the `Qt.createComponent()` function to create a component dynamically at runtime from within JavaScript, and then call `createObject()` to instantiate it.
- **Destroy Unused Elements**
  - Elements which are invisible because they are a child of a non-visible element (for example, the second tab in a tab-widget, while the first tab is shown) should be initialized lazily in most cases, and deleted when no longer in use, to avoid the ongoing cost of leaving them active (for example, rendering, animations, property binding evaluation, etc).
  - An item loaded with a Loader element may be released by resetting the "source" or "sourceComponent" property of the Loader, while other items may be explicitly released by calling `destroy()` on them. In some cases, it may be necessary to leave the item active, in which case it should be made invisible at the very least.

# Render

- **Over-drawing and Invisible Elements**
  - If you have elements which are totally covered by other (opaque) elements, it is best to set their "visible" property to false or they will be drawn needlessly.
- **Translucent vs Opaque**
  - Opaque content is generally a lot faster to draw than translucent. The reason being that translucent content needs blending and that the renderer can potentially optimize opaque content better.
- **Shader**
  - When deploying to low-end hardware and the shader is covering a large amount of pixels, one should keep the fragment shader to a few instructions to avoid poor performance.

# 文件管理器的优化

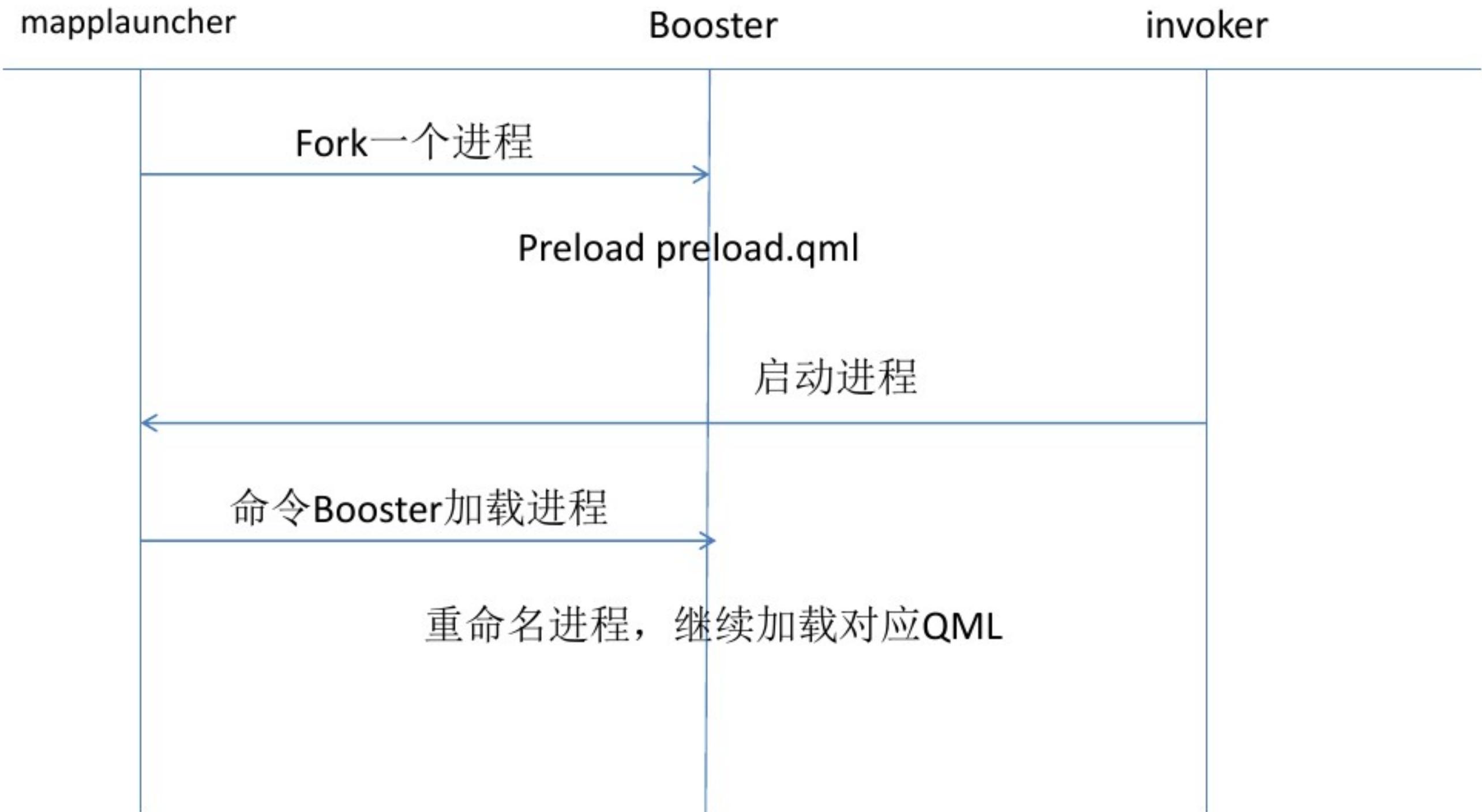
```
FileEditBar.qml{
    CMOS_BasicDialog {
        id:deleteConfirmDialog
    }
    CMOS_BasicDialog {
        id:renameDialog
    }
    CMOS_BasicDialog{
        id:progressDialog
    }
}
```

- 两种优化方法
  - 动态创建
  - Re-use existing components

# 应用启动的优化

- 生成Image时，运行prelink
- 使用Booster机制
- 进行CPU调频
- 调整应用的启动效果
- 发布Release版本

# Booster的机制



# 对Booster的优化

- 优化前

```
CMOS_PageStackWindow{  
    CMOS_BasicDialog{}  
    CMOSLineEdit {}  
    CMOSListView {}  
    CMOSTabView {}  
    CMOSPage {}  
}
```

- 优化后

```
Item{  
    Component{  
        Item{  
            CMOS_PageStackWindow{}  
            CMOS_BasicDialog {}  
            CMOSLineEdit {}  
            CMOSListView {}  
            CMOSTabView {}  
            CMOSPage {}  
        }  
    }  
}
```

# 调整应用启动的效果

- 为每个应用创建一个splash图片
- 启动应用的时候，显示启动动画，显示splash图片
  - 启动动画的时长从500ms，减少到300ms
- 在应用内容刷新后，在将splash与应用内容之间做个淡入淡出的效果

# 参考文档

- [qt-project.org/doc/qt-5/qtquick-performance.html](http://qt-project.org/doc/qt-5/qtquick-performance.html)
- Qml Optimization Tips

## QML Optimization tips

This is a checklist to go through when optimizing QML sources. Some of these points are general and some are from observations of going through existing QML sources. Notes are not in any specific order. If there are errors or some good notes missing, feel free to edit the document or contact Kaj Grönholm.

### 1. Use QML declaratively

Sometimes people are used to programming in imperative way with C/C++/JavaScript and try to use same with QML. This may result in slow performance and bugs. Instead try to study what kind of different QML elements are available and which properties and signals those elements provide. For example instead of using JavaScript with your own logic to disable MouseArea "if (myDisabled) {...", it provides "enabled" property.

### 2. Avoid deep hierarchies

Deep hierarchies of QML elements (e.g. Rectangle inside Item inside Image inside Rectangle inside...) should be avoided. Try to reduce the amount of elements by not introducing extra items when anchoring and margins are enough.

### 3. Limit JavaScript usage

JavaScript is baked into QML and it is OK to utilize it for simple UI logic. But JavaScript should not be used extensively, especially not in places which are called frequently like in animations.

### 4. Transparent Rectangle

Instead of using Rectangle with color set to "transparent" to get non-visible component, use Item as it is lighter and suitable for layouting UI elements

### 5. Use QtObject

Don't use Item for components which don't have any visual presentation and don't contain child components. Instead use QtObject which is lighter and doesn't increase the amount of items in Scene Graph. QtObject can contain id, properties and JavaScript functions.

### 6. Use Item as root element